HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

MASTERARBEIT

Herr
Eric Schubert

Optimierung von mobilen 3D-Anwendungen - Eine Machbarkeitsstudie

2015

Fakultät: Medien

MASTERARBEIT

Optimierung von mobilen 3D-Anwendungen - Eine Machbarkeitsstudie

Autor:
Herr Eric Schubert

Studiengang:
Information and Communication Science

Seminargruppe:
IC12w1-M

Erstprüfer:
Prof. Dr.-Ing. Robert J. Wierzbicki

Zweitprüfer:
Thomas Schmieder, M.A.

Einreichung:
Dresden, 15.12.15

Faculty of Media

MASTER THESIS

Optimization of mobile 3D-applications - a feasibility study

author:
Mr. Eric Schubert

course of studies:
Information and Communication Science

seminar group:
IC12w1-M

first examiner:
Prof. Dr.-Ing. Robert J. Wierzbicki

second examiner:
Thomas Schmieder, M.A.

submission:
Dresden, 15.12.15

Bibliografische Angaben

Schubert, Eric:

Optimierung von mobilen 3D-Anwendungen – Eine Machbarkeitsstudie

Optimization of mobile 3D-applications – a feasibility study

134 Seiten, Hochschule Mittweida, University of Applied Sciences,
Fakultät Medien, Masterarbeit, 2015

Abstract

Diese Masterarbeit behandelt die Optimierung von 3D-Anwendungen auf mobilen *Android*-Endgeräten. Die Optimierung erfolgt anhand der Leistungsanalyse von Smartphones und Tablets. Diese Arbeit umfasst die Erstellung eines Arbeitsablaufs, die Konzeption einer *Android*-App zur Leistungsanalyse sowie die exemplarische statistische Auswertung der aufgezeichneten Daten. Durch die statistische Auswertung der aufgezeichneten Daten können Flaschenhälse in Anwendungen analysiert und diese besser an spezifische Endgeräte angepasst werden. Dadurch können Produktionsprozesse für mobile 3D-Anwendungen optimiert werden.

Inhaltsverzeichnis

Abkürzungsverzeichnis

3D	dreidimensional
APK	Android Application Package
AR	Augmented Reality
ADB	Android Debug Bridge
Avg	Average
C#	C-Sharp
CSV	Comma Separated Values
DDMS	Dalvik Debug Monitor Server
Hz	Hertz
ms	Millisekunden
MB	Megabyte
CPU	Central Processing Unit
GPU	Graphics Processing Unit
PAL	Phase Alternating Line
PBR	Physically Based Rendering
QR	Quick Response
SDK	Software Development Kit
SoC	System on a Chip

Formelverzeichnis

FPS	Frames per second	[fps]
f	Frequenz	[Hz]
σ	Standardabweichung	
t	Zeit	[s]
r	Korrelationskoeffizient	

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Android

Betriebssystem für Smartphones und Tablets.

Android Application Package (APK)

Anwendungsdatei für das Android-Betriebssystem.

Augmented Reality (AR)

Unter Augmented Reality werden Anwendungen verstanden, die digitale Inhalte, in Abhängigkeit eines definierten Kontexts, in das Blickfeld des Betrachters einblenden. AR wird beispielsweise in Fußballübertragungen eingesetzt um Abseitslinien einzublenden.[1]

Batching

Sequentielles abarbeiten von Aufträgen zur Automatisierung.[2]

Bones

Virtuelle Knochen mit denen digitale Charaktere bewegt werden.

Buffer

Zwischenspeicher für Daten.

C#

An die Programmiersprache C++ angelegte Skriptsprache.[3]

---

[1] Vgl. TÖNNIS 2010, V
[2] Vgl. GERLICH 2005,494
[3] Vgl. WITTE 2004,15

Comma Separated Values (CSV)

Ein Dateiformat, das Daten in tabellarischer Form enthält. Einzelne Datenblöcke sind über Kommata voneinander getrennt. Mit Datensätzen in CSV-Form ist ein Datenaustausch zwischen Tabellenkalkulationsprogrammen und Statistikprogrammen auf unterschiedlichen Systemen möglich.[4]

Central Processing Unit (CPU)

Zentrale Berechnungseinheit eines Computers.[5]

Debugger

Anwendung zur Fehlerdiagnose. Über sie werden Fehlerquellen bestimmt und eliminiert.[6]

Drag and Drop

Steht für Ziehen und Ablegen. Hierdurch können Anwendungen gestartet werden, indem eine Datei auf ein Programmsymbol gezogen wird.[7]

Draw Calls

Sie signalisieren, wie oft die Grafikkarte bei einem ausgegeben Bild Inhalte berechnen muss.

Diffusemap

Bilddatei, welche die Hauptoberflächenfarbe eines 3D-Modells definiert.[8]

[4] Vgl. HETLAND 2008, 258
[5] Vgl. BIRN, 2007: 282
[6] Vgl. GERLICH, 2005: 485
[7] Vgl. BRUNKEN, 2015
[8] Vgl. BIRN, 2007: 288

Directional Light

Unendliches Licht ohne definierten Strahlbereich. Es beleuchtet eine Szene gleichmäßig aus einer Richtung.[9]

Game Engine

Eine Game Engine ist das Grundgerüst einer interaktiven 3D-Anwendung. Sie stellt Ein- und Ausgabeschnittstellen bereit.

Frame

Ein Einzelbild in der Computergrafik. Eine Aneinanderreihung von Frames erzeugt eine Animation.

Frame Rate

Die Anzahl der Bilder pro Sekunde (fps) werden als Frame Rate bezeichnet. Kinofilme werden mit 24 fps abgespielt. Bei 3D-Anwendungen ist die maximale Frame Rate abhängig vom verwendeten Monitor. Ein Monitor mit einer Bildwiederholrate von 60 Hz kann maximal 60 fps darstellen.

Eine flüssige Frame Rate ist subjektiv. Kinofilme werden mit 24 fps als flüssig empfunden, ebenso das PAL-Fernsehsignal mit 25 fps. Bei niedrigeren Frame Rates kann ein sichtbares Ruckeln auftreten.

Frame Time

Zeit, die für die Berechnung eines Frames benötigt wird.

[9] Vgl. FLAVELL; 2010: 78

Graphics Processing Unit (GPU)

Dedizierter Prozessor für die Berechnung von Echtzeitgrafiken von interaktiven Anwendungen und Spielen.[10] Dieser befindet sich entweder auf einer separaten Grafikkarte oder ist in den Prozessor oder in den Chipsatz der Hauptplatine integriert.

Integer

Ganzzahliger positiver oder negativer Wert.

iOS

Betriebssystem für Smartphones und Tablets von Apple.

Legacy

Beschreibt im Bereich der Informatik alten Code. In dieser Arbeit wird der Begriff im Zusammenhang mit alten Darstellungsmethoden in der Computergrafik genutzt.

Logfile

Datei, die aufgezeichnete Daten eines Prozesses enthält.

Normalmap

Bilddatei, die es erlaubt feine Oberflächendetails auf niedrig aufgelösten 3D-Modellen zu simulieren. Hierzu werden die Oberflächennormalen eines 3D-Modells auf Pixelbasis verändert. Der Winkel der Oberflächennormale wird über die drei Farbwerte der Bilddatei definiert.[11]

Mac OS

Betriebssystem von Apple.

---

[10] Vgl. BIRN, 2007: 282
[11] Vgl. BIRN, 2007: 297

Phase Alternating Line (PAL)

Verfahren zur Bildübertragung mit 25 Einzelbildern pro Sekunde.[12]

Physically Based Rendering (PBR)

Diese Methode der Bildsynthese basiert auf der natürlichen Energieerhaltung und ermöglicht besonders realistisch aussehende 3D-Grafiken.[13]

Vor der Einführung von PBR in Game-Engines war diese Methode in computeranimierten Filmen wie dem PiXAR-Film Toy Story 3 anzutreffen. Aufgrund der gesteigerten Leistungsfähigkeit moderner Computer und Spielekonsolen ist es möglich, diese Berechnungen in vereinfachter Form in Echtzeit durchzuführen.

Polygone

Bestehen aus mindestens drei Vertices und ermöglichen die Darstellung von Flächen im dreidimensionalen Raum. Polygone mit drei Eckpunkten werden als Triangle, bei vier Punkten als Quad und bei fünf oder mehr als n-Gon bezeichnet. N steht hierbei für eine beliebige Zahl.

Profiler

Ein Programm zur Leistungserfassung von Computerprorammen Über Profiler können Anwendungen analysiert und optimiert werden.[14]

Public

Öffentlich zugängliche Variable in der objektorientierten Programmierung.[15]

---

[12] Vgl. BIRN, 2007: 169
[13] Vgl. WILSON, http://www.marmoset.co/toolbag/learn/pbr-practice#faq, 01.11.15
[14] Vgl. SCHLIMM, 2007: 257
[15] Vgl. WITTE, 2004: 62

Python

Objektorientierte Programmiersprache. Zum Ausführen wird eine Python-Laufzeitumgebung benötigt.[16]

Quick Response-Code (QR)

Ein Bild, in dem digitale Informationen codiert sind. Mit Smartphones und Tablets kann auf die Inhalte dieser Bilder zugegriffen werden.

Rendern

Beschreibt die Umwandlung von 3D-Inhalten in ein zweidimensionales Bild, das auf dem Monitor abgebildet werden kann. Die dargestellten Inhalte können einfarbige Flächen oder komplexe 3D-Szenen mit aufwendiger Beleuchtung sein. Die Zeit, die für das Rendern eines Bildes benötigt wird ist von der Komplexität der 3D-Inhalte abhängig.

Shader

Sie definieren das optische Erscheinungsbild von Objekten. Über einen Shader wird festgelegt, wie sich ein Objekt unter Lichteinfall verhält. Beispielsweise gibt es reflektierende, matte, transparente und selbstleuchtende Shader.[17]

Skript

Im Gegensatz zu Computeranwendungen, die eigenständig Lauffähig sind, müssen Skripte in einer Laufzeitumgebung ausgeführt werden. Beispiele für Skriptsprachen sind Python und C#.

Skydome

Eine Lichtquelle, die eine Umgebung von allen Seiten beleuchtet. Über ein Skylight wird das in der Atmosphäre gestreute Sonnenlicht simuliert[18].

---

[16] Vgl. HETLAND, 2008: xxix
[17] Vgl. BIRN, 2007: 252
[18] Vgl. SMITH, 2006: 299

System on a Chip (SoC)

Mikroprozessor von Mobilgeräten der die Funktionen vieler Chips in sich vereint. In ihm sind beispielsweise optische, digitale oder thermische Elemente zusammengefasst.[19]

Software Development Kit (SDK)

Entwicklungsumgebung für Anwendungen.

Texturen

Sie werden auf die Oberfläche von dreidimensionalen Körpern projiziert, um ihnen Farben und Strukturen zu verleihen.[20]

Triangles

Aufgrund der Funktionsweise von Computergrafikbeschleunigern werden innerhalb von Echtzeitanwendungen nur dreieckige Polygone verarbeitet. [21]

Vertices

Sie sind die Eckpunkte eines 3D-Objektes. Mehrere Vertices definieren Triangles oder Polygone.[22]

Windows Phone

Mobiles Betriebssystem von Microsoft.

[19] Vgl. REIS, 2006: 2
[20] Vgl. NISCHWITZ, 2011: 266
[21] Vgl. MALIZIA, 2006: 15
[22] Vgl. GREENBERG, 2007: 698

1 Einleitung

Auf mobilen Endgeräten ausgeführte 3D-Anwendungen spielen für Firmen eine immer wichtigere Rolle. Unternehmen wie die Brandstätter-Gruppe, zu dem die Marke *PLAYMOBIL* gehört, haben für Marketingzwecke ihre aktuellen Produktkataloge mit *QR-Codes* versehen.[23] Diese können über Smartphones und Tablets mit einer App gelesen werden. Mithilfe von *Augmented Reality* werden die Kataloge zum Leben erweckt und 3D-Umgebungen verschmelzen mit der realen Welt.

Die Qualität der in diesen Anwendungen dargestellten 3D-Umgebungen hängt von der Leistungsfähigkeit der Endgeräte ab. Die Leistung dieser Geräte wird in Prospekten und Katalogen mit Begriffen und Zahlen wie Gigahertz, Gigabyte und der Anzahl an Prozessorkernen definiert. Diese Werte geben jedoch keinen Rückschluss auf die konkrete Hardwareleistung in bestimmten Szenarien und werden oft nur genutzt, um neue Geräte zu bewerben. Von diesen Werten lässt sich somit kein Rückschluss auf die eigentliche 3D-Leistung eines Endgerätes geben. Von der 3D-Leistung hängt ab, bei welchem Detailgrad der 3D-Umgebungen die Grafiken noch, für das menschliche Auge, flüssig dargestellt werden können.

Diese Masterarbeit wird für *PiXABLE STUDIOS*[24] geschrieben. Das Dresdner Animationsstudio existiert seit 2005 und wurde 2014 von der *Mastersolution AG* aus Plauen übernommen. *PiXABLE* produziert computeranimierte Fernsehspots und 3D-Umgebungen für Videospiele.[25] Zu den von *PiXABLE STUDIOS* erstellten mobilen 3D-Anwendungen gehören unter anderem die die 3D-Umgebungen der *AR*-App für den *PLAYMOBIL*-Katalog.

1.1 Zielsetzung

Ziel dieser Masterarbeit ist es, Leistungslimitationen von mobilen Endgeräten frühzeitig zu erkennen um den Prozess der Erstellung von mobilen 3D-Anwendungen für *PiXABLE STUDIOS* zu beschleunigen. Bei den Leistungslimitationen geht es um jene, welche die Darstellung von 3D-Umgebungen auf bestimmten Endgeräten negativ beeinflussen.

---

[23] https://play.google.com/store/apps/details?id=com.playmobil.pmscan&hl=de, 05.10.15

[24] Vgl. PiXABLE STUDIOS, http://www.pixablestudios.com/, 01.11.15

[25] Vgl. MASTERSOLUTION, http://blog.mastersolution.ag/2014/03/06/mastersolution-ag-ubernimmt-pixable-studios/, 01.11.15

Hierfür soll ein Analyseprozess erarbeitet werden, der eine statistische Auswertung von mobilen Endgeräten ermöglicht. Auf Basis statistischer Daten soll überprüft werden, ob Faktoren identifiziert werden können, die sich negativ, auf die 3D-Leistung auswirken. Mithilfe solcher wäre es möglich, eine App gezielt auf bestimmte mobile Endgeräte hin zu optimieren. Neben Apps, die für Endkunden gedacht sind, entwickelt *PiXABLE* Apps die bei verschiedenen Unternehmen intern eingesetzt werden. Diese Apps laufen auf spezifischen Geräten, entsprechend müssen die Apps für diese Geräte optimiert werden. Für dieses Vorhaben muss eine App generiert werden, mit welcher Leistungsdaten von mobilen Endgeräten aufgezeichnet werden können. In dieser Arbeit wird auf den Entwicklungsprozess sowie den Aufbau einer solchen App eingegangen.

Apps für die Leistungserfassung mobiler Endgeräte gibt es bereits. Diese, beispielsweise *3DMARK*[26] von *Futuremark*[27], arbeiten mit einem Punktesystem um die Leistung der Hardware zu bewerten. Die Punktzahl wird über die *Frame Rate* ermittelt, die ein Gerät in bestimmten Tests und Zeitabschnitten erreicht hat. Die Ergebnisse der Tests können abschließend über die *Futuremark*-Internetseite[28] miteinander verglichen werden. Ein Ziel der Arbeit ist es, zu prüfen, ob Daten ähnlich wie bei der *Futuremark*-App über eine Verteilung der Analyseanwendung gesammelt werden können. Ein Nachteil der *Futuremark*-Anwendung ist, dass Parameter, welche sich negativ auf die Anwendungsleistung und entsprechend auf die *Frame Rate* auswirken, über dieses Punktesystem nicht identifiziert werden können. Um eine Anwendung auf bestimmte Endgeräte hin zu optimieren, ist es jedoch wichtig diese, mithilfe von detaillierten Leistungsdaten, zu ermitteln. Diese Faktoren können beispielsweise bestimmte Berechnungsarten von 3D-Umgebungen oder *Shader* sein.

1.2 Zielgruppe

Diese Masterarbeit richtet sich an App-Entwickler. Es werden Grundkenntnisse im Bereich der Programmierung und 3D-Grafik vorausgesetzt. Es wird nicht näher auf die Entwicklung von 3D-Umgebungen und Apps eingegangen. Aus Gründen der besseren Lesbarkeit werden viele englische Fachbegriffe verwendet. Innerhalb dieser Arbeit wurden *Skripte* in *Python* und *C#* geschrieben. Die Funktionsweise dieser *Skripte* wird in den jeweiligen Kapiteln beschrieben. Es wird jedoch nicht im Detail auf die verwendeten

[26] FUTUREMARK, http://www.futuremark.com/benchmarks/3dmark/all, 31.10.15
[27] FUTUREMARK, http://www.futuremark.com, 31.10.15
[28] FUTUREMARK, http://www.futuremark.com/hardware/mobile, 31.10.15

Programmierbefehle eingegangen. Nähere Informationen zu den einzelnen Befehlen können in den *Python*- und *C#*-Dokumentationen nachgelesen werden.

1.3 Vorgehensweise

Die Arbeit gliedert sich in vier Teilbereiche. Im ersten Abschnitt wird ein Arbeitsablauf zur Leistungsanalyse erstellt. Im zweiten Abschnitt der Arbeit wird geprüft, ob die aufgezeichneten Daten statistisch relevant sind. Es wird analysiert, durch welche Faktoren die Leistungsanalyse beeinflusst wird und ob bei wiederholter Aufzeichnung der Leistungsdaten ähnliche Ergebnisse entstehen. Der dritte Teil der Arbeit widmet sich der Erstellung einer Analyseanwendung anhand vordefinierter Forschungsfragen. Hierfür wird eine App generiert, mit der die Forschungsfragen geprüft werden. Im letzten Teil werden die statistischen Daten, welche aus der Analyseanwendung gewonnen wurden, anhand der Forschungsfragen ausgewertet.

Für die Erstellung der mobilen 3D-Umgebungen für diese Masterarbeit wird *Unity* genutzt. *PiXABLE STUDIOS* generierten mithilfe von *Unity* die 3D-Umgebungen für die *PLAYMOBIL Scan*[29] App. *Unity* besteht seit 2005[30] und ist in der Basisversion für Privatanwender und Unternehmen kostenlos. Abbildung 1 zeigt, dass *Unity* bei Entwicklern von Anwendungen sehr beliebt ist. Nutzer von *Unity* sind beispielsweise *Electronic Arts*, *LEGO* oder die *NASA*.[31]

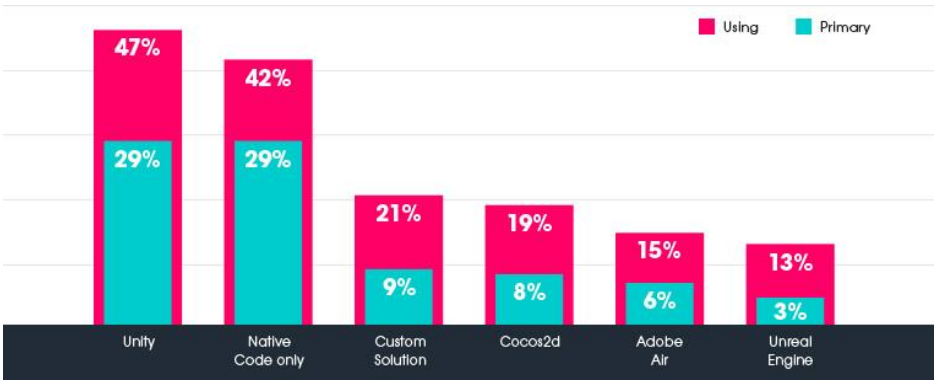

Abbildung 1: Verbreitung von Game Engines[32]

[29] https://play.google.com/store/apps/details?id=com.playmobil.pmscan&hl=de, 13.10.15
[30] Vgl. UNITY3D, https://unity3d.com/10-year-anniversary, 31.08.15
[31] Vgl. UNITY3D, https://unity3d.com/public-relations, 15.10.15
[32] Vgl. Ebenda

Unity stellt eine Entwicklungsumgebung und eine *Game Engine* für 3D-Anwendungen bereit.[33] Über den *Unity Editor* ist es möglich Videospiele für eine große Anzahl von Plattformen zu entwickeln. Das Spieleportfolio umfasst Indie-Games, wie unter anderem *Kerbal Space Program*.[34] Etablierte Entwickler nutzen ebenfalls Unity für Ihre Spiele, ein Beispiel hierfür ist *Cities Skylines* von *Paradox Interactive*.[35]

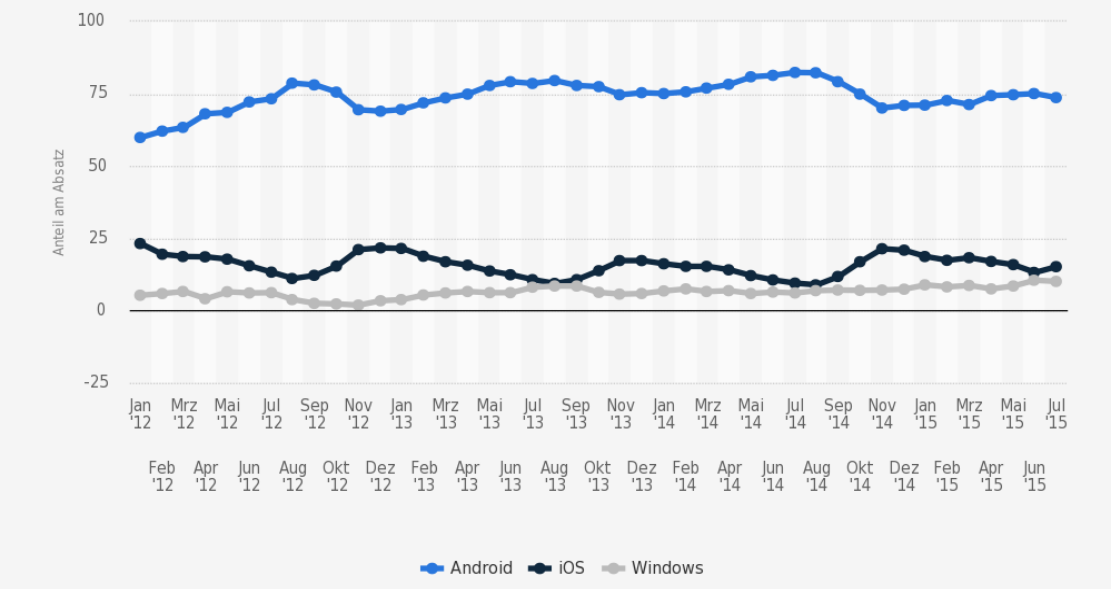

Abbildung 2: Marktanteile der mobilen Betriebssysteme am Absatz von Smartphones in Deutschland von Januar 2012 bis Juli 2015[36]

Im Rahmen dieser Masterarbeit wird auf die Leistungsanalyse von Mobilgeräten mit *Android*-Betriebssystem eingegangen. Eine Leistungserfassung von *iOS*- und *Windows Phone*-Geräten ist mit *Unity* ebenfalls möglich. Aufgrund des in Abbildung 2 erkennbar niedrigen Marktanteils von 10,5 %, wird auf einen Test mit *Windows Phone*-Geräten verzichtet. Geräte mit *iOS*, kommen auf einen minimal höheren Marktanteil, werden aber ebenfalls nicht getestet, da eine Entwicklung von *iOS*-Anwendungen nur auf einem Apple-Computer mit *Mac OS* möglich ist.[37] Über diese Hardware verfügt *PiXABLE STUDIOS* nicht.

---

[33] Vgl. CHIN, 2010: 192

[34] Vgl. UNITY3D, https://unity3d.com/showcase/gallery, 31.08.15

[35] Ebenda

[36] Vgl. STATISTA, http://de.statista.com/statistik/daten/studie/225381/umfrage/marktanteile-der-betriebssysteme-am-smartphone-absatz-in-deutschland-zeitreihe/, 31.08.15

[37] Vgl. APPLE DEVELOPER, 7

Für die Leistungsanalyse mobiler Endgeräte wird mit *Unity* eine App generiert die aus einer Aneinanderreihung von Testszenen besteht. Jede dieser Szenen prüft verschiedene, für die Leistungsfähigkeit relevante, Parameter. Beispielsweise wird durch die Berechnung unterschiedlicher Charakterzahlen die Leistungsfähigkeit bei der Berechnung von bewegten, sich deformierenden Objekten aufgezeichnet. Die Anzahl der verwendeten Charaktere wird sich hierbei mit der Zeit stetig erhöhen, um ausreichend Datensätze zu liefern. Es kommen ebenfalls verschieden detaillierte 3D-Umgebungen zum Einsatz, um die Auswirkungen von höher aufgelösten 3D-Objekten zu ermitteln.

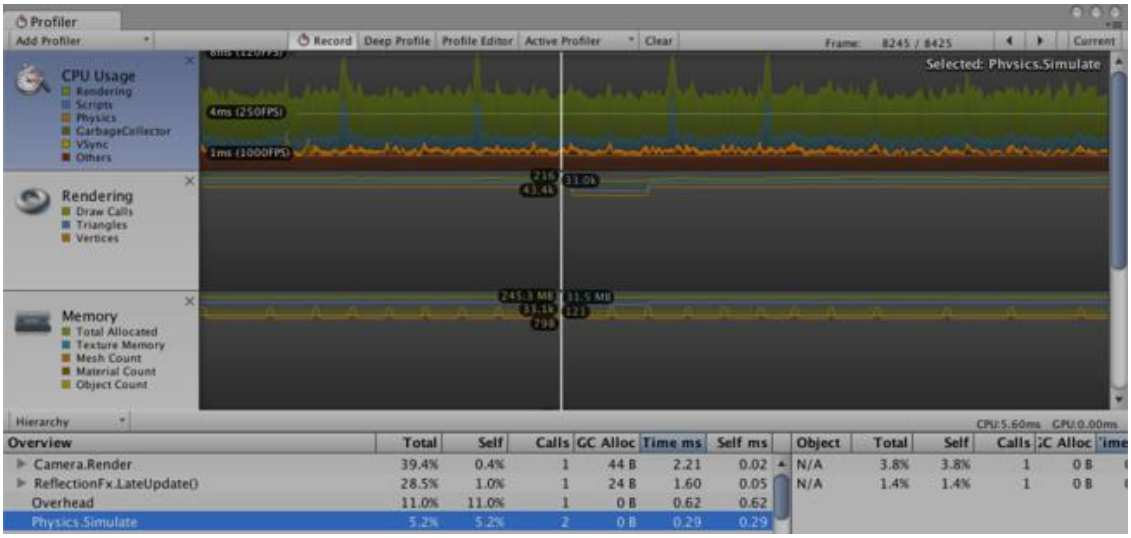

Abbildung 3: Profiler im Unity-Editor[38]

Die Leistungsanalyse der Endgeräte erfolgt mithilfe des in *Unity* integrierten *Profilers*. Dieses Werkzeug ermöglicht es Leistungsdaten einer 3D-Anwendung zu analysieren.[39]

Es gibt zwei Varianten, den *Unity-Profiler* zu verwenden. Der *Profiler* kann, wie in Abbildung 3, direkt im *Unity*-Editor zur Analyse einer App genutzt werden. Die Leistungsdaten werden in Echtzeit angezeigt, können jedoch nicht gespeichert werden. Die zweite Variante, welche in dieser Arbeit verwendet wird, ist die Aufzeichnung der Leistungsdaten über eine *Debug*-Schnittstelle. Hierbei werden die Rohdaten, die vom *Unity-Profiler* gesendet werden, direkt abgegriffen. Diese Variante ermöglicht es die Leistungsfähigkeit einer App auch ohne eine laufende Instanz von *Unity* zu auszuwerten.

[38] UNITY, http://docs.unity3d.com/uploads/Main/ProfilerTimeline.png, 10.11.15

[39] Vgl. UNITY, http://docs.unity3d.com/Manual/Profiler.html, 05.10.15

Folgende Liste zeigt einen Ausschnitt der Rohdaten des *Profilers*, für dem Zeitrahmen von einer Sekunde.

3scenes_125seconds_01.txt[40]

```
07-22 10:26:22.169: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:22.169: D/Unity(14094): cpu-player>    min:  0.0   max: 27.9  (…)
07-22 10:26:22.169: D/Unity(14094): cpu-ogles-drv> min:  0.0   max:  0.0   (…)
07-22 10:26:22.169: D/Unity(14094): cpu-present>   min:  0.0   max:  0.9   (…)
07-22 10:26:22.169: D/Unity(14094): frametime>     min:  0.0   max: 28.0    (…)
07-22 10:26:22.169: D/Unity(14094): batches>       min:   0    max:   4    avg:   3
07-22 10:26:22.169: D/Unity(14094): draw calls>    min:   0    max:   4    avg:   3
07-22 10:26:22.169: D/Unity(14094): tris>          min:    0  max:  3218  avg:  3164
07-22 10:26:22.169: D/Unity(14094): verts>         min:    0  max:  6074  avg:  5972
07-22 10:26:22.169: D/Unity(14094): dynamic batching> batched draw calls: (…)
07-22 10:26:22.169: D/Unity(14094): static batching>  batched draw calls: (…)
07-22 10:26:22.169: D/Unity(14094): player-detail> physx:  0.6 animation: (…)
07-22 10:26:22.169: D/Unity(14094): mono-scripts>  update:  0.0    (…)
07-22 10:26:22.169: D/Unity(14094): mono-memory>   used heap: 31948 (…)
```

Die auf diesem Weg über den *Profiler* ermittelten Daten sollen aufgezeichnet und tabellarisch erfasst werden. Die Erfassung der Daten soll möglichst automatisiert ablaufen. Die Aufbereitung der Daten, um eine statistische Auswertung zu ermöglichen, wird im Rahmen dieser Masterarbeit ausführlich beschrieben. Mit den aufbereiteten Daten werden Forschungsfragen zur Leistungsfähigkeit mobiler Endgeräte geprüft. Hierzu wird auf Datensätze verschiedener analysierter Geräte zurückgegriffen.

[40] Anlage CD /3scenes_125seconds/3scenes_125seconds_01.txt

2 Test zur Umsetzung einer Leistungsanalyse

In diesem Kapitel wird überprüft, ob eine Leistungserfassung mithilfe von *Unity* auf *Android*-Geräten möglich ist. Hierfür wird zunächst ein Analyseaufbau beschrieben mit dem eine Leistungsanalyse möglich ist. Anhand dieses Aufbaus wurden Vorabtests mit einer dafür erstellten *Unity*-App durchgeführt. Innerhalb dieser Tests wurde überprüft, wie eine Leistungsanalyse durchgeführt werden kann. Außerdem wurde getestet, ob eine *Unity* 3D-Anwendung bei mehreren Durchläufen auf einem Testgerät gleichbleibende Ergebnisse ermöglicht. Objekte und Charaktere und Technik für die Testanwendung wurden von *PiXABLE STUDIOS* bereitgestellt.

Die Leistungsanalyse von Android-Geräten läuft folgendermaßen ab. Eine App wird mit *Unity* generiert und auf ein beliebiges *Android*-Gerät übertragen. Während über eine *Debug*-Schnittstelle eine Datenverbindung zu einem PC besteht, wird die App ausgeführt. Über ein Analyseprogramm werden die Leistungsdaten der App aufgezeichnet. Nachdem die App durchlaufen ist, werden die Ergebnisse in Tabellenform abgespeichert und statistisch analysiert.

2.1 Aufbau der Testumgebung

Um *Android*-Anwendungen zu generieren und zu *debuggen*, werden Anforderungen an die Hard- und Software gestellt. Neben *Unity* und einem *Android*-Gerät wird zur Durchführung einer Leistungsanalyse eine Reihe von Anwendungen sowie Einstellungen auf den Testgeräten sowie dem für die Aufzeichnung angeschlossenen PC benötigt.[41] Im folgenden Abschnitt wird näher auf die verwendeten Aufzeichnungssysteme und benötigte Software eingegangen.

[41] Vgl. ANDROID DEVELOPER, http://developer.android.com/tools/debugging/index.html, 11.10.15

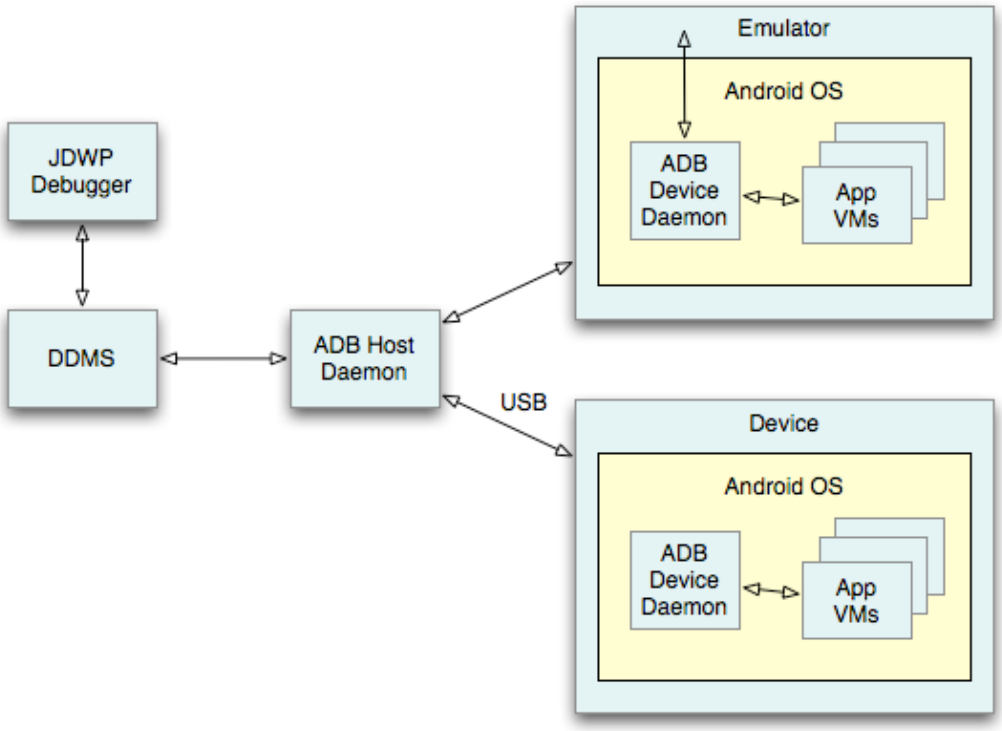

Abbildung 4: Android-Debugging-Aufbau[42]

Abbildung 4 zeigt den für das *Debugging* von *Android*-Geräten benötigten Versuchsaufbau. Innerhalb dieser Masterarbeit wird das *Debugging* von *Android*-Geräten mithilfe des *Dalvik Debug Monitor Service* (*DDMS*) [43] über USB durchgeführt. Als Softwareschnittstelle zwischen dem Gerät und dem PC dient die *Android Debug Bridge*[44] (*ADB*). Sie ermöglicht die Kommunikation zwischen den *Android*-Gerät und einem Host, in diesem Fall *DDMS*.[45] Zur Leistungserfassung muss das zu testende *Android*-Gerät mit einem PC verbunden sein. Die Verbindung zwischen *Android*-Gerät und PC erfolgt über ein USB-Kabel. Ein *Debugging* ist ebenfalls über WLAN möglich.[46] Hierzu müssen das *Android*-Gerät und der für die Aufzeichnung verwendete PC im selben WLAN-Netzwerk sein. Die auf dem *Android*–Gerät laufende App wurde mit *Unity* in der Version 5.1.0f3[47] erstellt. Auf die in Abbildung 4 gezeigten Programme wird in diesem Unterkapitel genauer eingegangen

[42] ANDROID DEVELOPER, http://developer.android.com/images/debugging.png, 11.10.15
[43] Vgl. ANDROID DEVELOPER, https://developer.android.com/tools/debugging/ddms.html, 31.08.15
[44] Vgl. ANDROID DEVELOPER, https://developer.android.com/tools/help/adb.html, 31.08.15
[45] Vgl. KRAJCI, 2014 :139
[46] Vgl. DROIDWIKI, https://www.droidwiki.de/Wireless_ADB, 31.10.15
[47] UNITY3D, https://unity3d.com/get-unity/download/archive, 31.08.15

Folgende Programme und Computersysteme wurden bei der Leistungsanalyse verwendet.

Android SDK

Um die *Unity* 3D-Anwendung für *Android*-Geräte zu kompilieren, muss das *Android SDK*[48] auf dem PC installiert sein. Dieses *SDK* stellt zudem den zur Aufnahme der *Profiler*-Daten relevanten *Dalvik Debug Monitor Service* (DDMS) bereit. Dieser wird zum empfangen *Unity-Profiler*-Daten vom *Android*-Gerät über die *Android Debug Bridge* (ADB) benötigt.

Java

Da *Android*-Apps auf Java basieren, wird für deren Erstellung eine installierte Java-Laufzeitumgebung[49] benötigt. Auf Computern, die lediglich zur Aufzeichnung der statistischen Daten genutzt werden, ist Java nicht notwendig.

Python

Laufzeitumgebung, die zum Ausführen von *Python-Skripten* benötigt wird. Für diese Masterarbeit wurde *Python* in der Version 2.7[50] verwendet. Mittels *Python* werden die aus *Unity* generierten Leistungsdaten in ein für die statistische Auswertung lesbares Format umgewandelt. Der Umwandlungsprozess wird in Kapitel 3.6 genauer beschrieben. Zur Aufzeichnung der Leistungsdaten wird *Python* nicht benötigt.

Computersysteme

Die Aufzeichnungen der Leistungsdaten wurden auf verschiedenen Computern mit den Betriebssystemen Windows 7, Windows 8.1 und Windows 10 in der 64-Bit-Version durchgeführt. In Tabelle 1 sind alle für den Test verwendeten Computer aufgelistet. Die Computerleistung reicht von einer modernen Workstation, über einen Mittelklasse-PC

[48] ANDROID DEVELOPER, https://developer.android.com/sdk/index.html, 31.08.15
[49] JAVA, https://www.java.com/de/download/, 31.08.15
[50] PYTHON, https://www.python.org/download/releases/2.7/ , 31.08.15

bis zu einem 8 Jahre alten Notebook. In Kapitel 3.4 wird geprüft, ob durch die Verwendung verschiedener Computer zur Aufzeichnung der Leistungsdaten die Ergebnisse beeinflusst werden.

	System 1	**System 2**	**System 3**
Prozessor	Intel Core i7 2600	Intel Core i3 3220	Intel T8100
Arbeitsspeicher	16,0 GB	8,0 GB	4,0 GB
Grafikkarte	nVidia Quadro 4000	nVidia geForce 750	Intel X3100
Betriebssystem	Windows 7, 64 Bit	Windows 10, 64 Bit	Windows 8.1, 64 Bit

Tabelle 1: Testsysteme zur Aufzeichnung der Testanwendung

Anforderungen an die Android-Geräte

Für die in diesem Kapitel durchgeführten Tests wurde ein *Samsung Galaxy S3* mit *Android* 4.4.4 verwendet. Um ein reales Anwendungsszenario abzubilden, wurde das Gerät für die Tests nicht bereinigt. Für jedes zu testende *Android*-Gerät müssen ebenfalls bestimmte Einstellungen getätigt werden, diese werden im folgenden Absatz beschrieben.

Abhängig vom zu testenden *Android*-Gerät muss zusätzlich ein passender USB-Treiber zur Verbindung mit dem PC vorhanden sein. Für das genutzte Samsung Smartphone wurde der Treiber direkt von der Samsung-Webseite bezogen.[51] Um die Daten des *Unity-Profilers* an einem PC aufzuzeichnen muss, bei jedem zu testenden *Android*-Gerät USB-*Debugging* aktiv sein. Dies ist Voraussetzung für das Senden von *Android-Profiler*-Daten über USB. Ab *Android* 4.2 ist USB-*Debugging* erst verfügbar, wenn der Nutzer

[51] SAMSUNG DEVELOPER, http://developer.samsung.com/technical-doc/view.do?v=T000000117, 05.10.15

siebenmal in den Einstellungen auf die aktuelle Betriebssystemversion getippt hat. Im Anschluss daran kann USB-*Debugging* aktiviert werden.[52]

Seitens der mit *Unity* erstellten App ist es wichtig, dass diese auf dem *Android*-Gerät ebenfalls im Entwicklermodus läuft. Nur in diesem Modus ist der *Profiler* aktiv und überträgt die Leistungsdaten via *ADB* an den PC. Die Daten können mittels *DDMS* empfangen werden. Die Aktivierung des *Unity-Profilers* hat eine minimale Auswirkung auf die Anwendungsleistung:

> *"Note that profiling has to instrument your code. This instrumentation has a small impact on the performance of your game. Typically this overhead is small enough to not affect the game framerate."*[53]

Die mit *DDMS* aufgezeichneten Daten lassen sich als Textdatei auf dem PC speichern. Diese Textdatei enthält alle vom *Android*-Gerät ausgehenden Daten.

2.2 Fragenkatalog zur Testanwendung

Um zu prüfen, ob eine Anwendung zur Leistungserfassung möglich ist, wurde ein Fragenkatalog zusammengestellt. Diese Fragen werden mithilfe einer mit *Unity* erstellten *Android*-App geprüft.

Ist der Aufzeichnungsprozess reproduzierbar?

Um unterschiedliche Geräte vergleichen zu können, müssen die Zeiten der aufgenommenen Daten identisch sein. Es muss dementsprechend überprüft werden, ob wiederholte Tests unter gleichen Bedingungen ähnliche Ergebnisse bringen.

Beeinflusst die Länge der Testanwendung die aufgenommenen Daten?

Es ist zu erwarten, dass viele Daten vom *Android*-Gerät zu dem PC übertragen werden. Infolgedessen muss überprüft werden, ob dies die Daten verfälscht oder es sogar zu Datenverlusten kommen kann.

---

[52] Vgl. ANDROID DEVELOPER, http://developer.android.com/tools/help/adb.html, 31.10.15

[53] UNITY, http://docs.unity3d.com/Manual/Profiler.html, 31.08.15

Sind die Testdaten statistisch auswertbar?

Hierfür muss geklärt werden, ob die Rohdaten statistisch verwertbar sind. Ist dies nicht der Fall, muss ein Weg gefunden werden, die Leistungsdaten so aufzubereiten, dass eine Auswertung möglich ist.

Beeinflussen die Aufzeichnungssysteme die Daten?

Es muss geprüft werden, ob unterschiedliche Aufzeichnungssysteme in gleichen Datensätzen resultieren.

Ist ein gesteuerter Wechsel von Szenen und Objekten möglich?

Mit einem automatisierten Wechsel von Objekten und Szenen kann der Analyseablauf vereinfacht werden. Der automatisierte Wechsel soll es ermöglichen Objekte und Umgebungen nacheinander zu laden. Ohne diese Funktionalität müsste für jede zu ermittelnde Situation eine neue Leistungsaufnahme gestartet werden.

Ist eine Verwendung der PBR-Shader auf mobilen Endgeräten möglich?

Es soll geprüft werden, ob die mit *Unity* 5 eingeführten *PBR-Shader* auf mobilen Geräten lauffähig sind. Ist dies nicht der Fall, muss auf die *Legacy-Shader* zurückgegriffen werden.

Anhand dieser Testfragen wurde eine App erstellt, welche diese klären sollte. Für den Test wurde eine App erstellt, die aus drei *Unity* 3D-Umgebungen besteht. Die Gesamtlänge der Anwendung beträgt 125 Sekunden. Die drei Teilbereiche der Testanwendungen bestehen aus unterschiedlichen Szenarien. Im ersten Szenario wird eine Reihe von simplen 3D-Geometrien angezeigt. Im zweiten werden immer komplexer werdende 3D-Charaktermodelle geladen und im letzten Abschnitt wird eine komplette 3D-Umgebung der *PLAYMOBIL*-App dargestellt. Auf den genauen Aufbau der App wird in Kapitel 2.3 eingegangen.

2.3 Aufbau der Testanwendung

In den folgenden vier Unterkapiteln wird auf die verwendeten *Skripte* und 3D-Umgebungen eingegangen.

2.3.1 Verwendete Skripte

Um den Objekt und Szenenwechsel sowie das Beenden der Anwendung zu ermöglichen, wurde auf die *Skriptsprache C#* zurückgegriffen. Alle verwendeten *Skripte* nutzen extern erreichbare Parameter. All diese sind in den *Skripten* als *Public* deklariert. Diese Parameter ermöglichen das Ändern von Zeitgebern oder Objektzahlen ohne den *Skript*-Quellcode, für jede Situation, anpassen zu müssen. Dies erlaubt die Verwendung des *Skript*s in verschiedenen *Unity*-Szenen. Abbildung 5 zeigt externe-Parameter von zwei der verwendeten *C#-Skripte* im *Unity*-Editor.

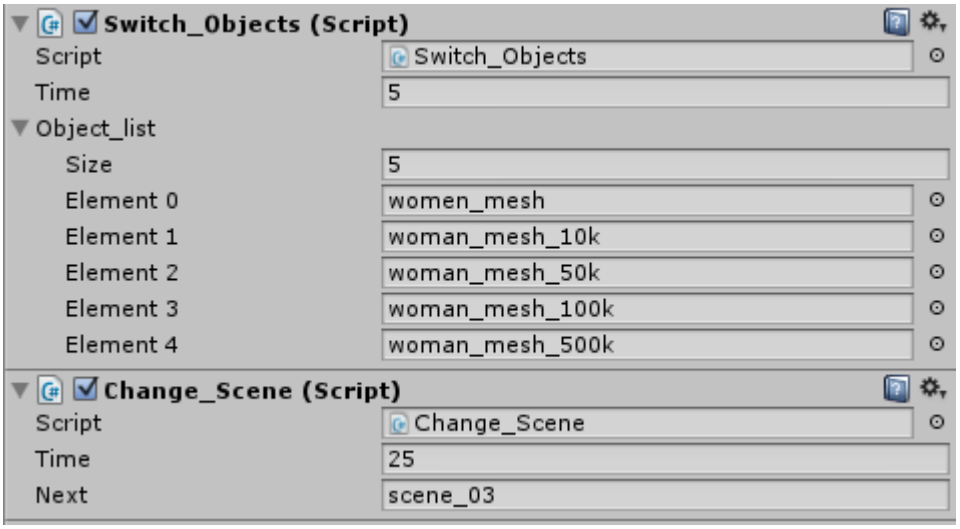

Abbildung 5: Externe Parameter der C#-Skripte

Steuerung des Szenenwechsels

Das *Change_Scene.cs-Skript* für den Szenenwechsel in Anlage 1 wird mit einem Zeitgeber gesteuert. Dieser legt fest, nach wieviel Sekunden der Wechsel in die nächste Testszene erfolgt. Der Zeitpunkt und die Folgeszene sind jeweils über einen *Public*-Parameter deklariert und können für jede Szene individuell angepasst werden. Standardmäßig ist dieser Parameter leer. Damit das *Skript* funktioniert, muss der Name einer Szene in das *Skript* eingetragen werden.

Beenden der App

Auf die gleiche Art wie der Szenenwechsel wird das Ende der App gesteuert. Im *Quit.cs Skript* in Anlage 8 gibt es einen Zeitgeber, welcher beim Erreichen einer vorher definierten Zeit die Anwendung beendet.

Steuerung des Objektwechsels

Das *Skript* mit dem Namen *Switch_Objects.cs* in Anlage 9 verfügt ebenfalls über einen Zeitgeber. Dieses *Skript* ruft nacheinander unterschiedliche Objekte auf. Diese Objekte befinden sich in einer vordefinierten Liste. Über einen Parameter kann die Zeit definiert werden, für die ein Objekt angezeigt wird. Die Zeit die für den Durchlauf des *Skript*es benötigt wird, ist abhängig von der Anzahl der Objekte in der Liste. Die im *Skript* definierte Zeit ist die Anzeigedauer für ein Listenelement.

Innerhalb des *Skript*es wird für jedes Element eine Schleife durchlaufen. Innerhalb dieser wird ein in der Liste eingetragenes Objekt aktiviert. Nachdem der Zeitgeber abgelaufen ist, wird dieses wieder deaktiviert. Dieser Vorgang wird so oft wiederholt, bis alle Listenelemente durchlaufen sind. Zur besseren Auswertung wird über einen *Debug*-Befehl der Hinweis des Objektwechsels ausgegeben.

2.3.2 Szene 1

Testszene 1 besteht aus 5 geometrischen Grundobjekten. Diese sind einfache 3D-Sphären. Diese Objekte sind in einer Reihe angeordnet und zu Beginn der Szene inaktiv. Die 5 Objekte werden von einem *Directional Light* und einem *Skydome* beleuchtet. Das *Directional Light* fungiert als Sonnenlicht während der *Skydome* die Schattierten Bereiche der Szenerie aufhellt. Hierdurch wird eine Beleuchtung ähnlich des Tageslichts simuliert.

Betrachtet wird die Szene durch eine Kamera, die aus einer Vogelperspektive auf die 5 Sphären herabblickt. Abbildung 6 zeigt den Aufbau der ersten Szene. Zur besseren Visualisierung der Szene wurden alle 5 Sphären eingeblendet.

Jedes Material der dreiteiligen Testanwendung verwendet die *PBR-Shader*. Durch diese leistungsintensiveren *Shader* wird eine höhere Systemlast erzeugt. Hierdurch wird geprüft, ob eine Verwendung dieser *Shader* für die spätere Anwendung in Betracht gezogen werden kann oder ob Abstürze auftreten.

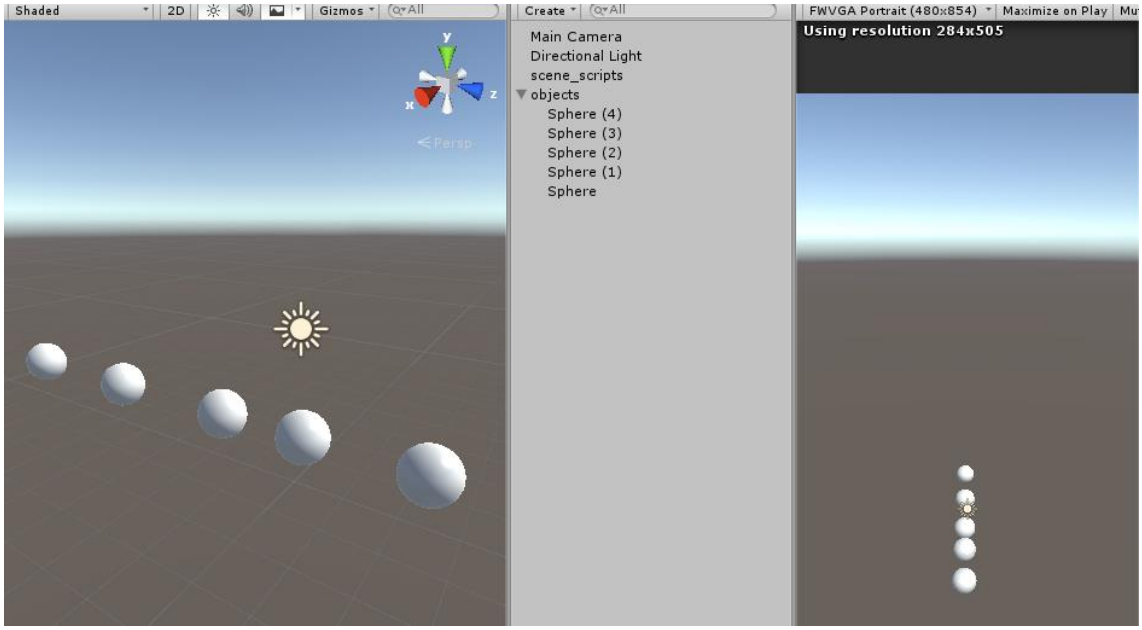

Abbildung 6: Szene 1 der Testanwendung mit eingeblendeten Sphären

Der Ablauf der Szene wird über ein Objekt mit dem Namen *scene_scripts* gesteuert. Auf diesem Objekt befinden sich alle für die Ausführung der Szene relevanten *Skripte*. Über das *Change_Objects.cs-Skript* werden die 5 Sphären nacheinander eingeblendet. Neben dem Change_*Objects.cs-Skript* befindet sich das *Change_Scene.cs-Skript* auf dem Objekt. Die Szenendauer beträgt insgesamt 75 Sekunden. Jede der Sphären wird für 15 Sekunden angezeigt. Nach Ablauf der 15 Sekunden wird das aktuelle Objekt ausgeblendet und das nächste Objekt wird sichtbar. Sobald der Zeitgeber des *Change_Scene.cs-Skriptes* bei 75 Sekunden angekommen ist, wird der Wechsel zur nächsten Testszene mit der Bezeichnung *scene_02* durchgeführt.

2.3.3 Szene 2

Der Aufbau der zweiten Testszene entspricht weitestgehend dem von Szene 1. Im Gegensatz zu 5 Sphären werden in dieser jedoch Charaktermodelle ein- und ausgeblendet. Die verwendete Beleuchtung, die Kamera sowie die Szenenhierarchie sind exakt dieselbe. Abbildung 7 zeigt den Aufbau von Szene 2 der Testanwendung.

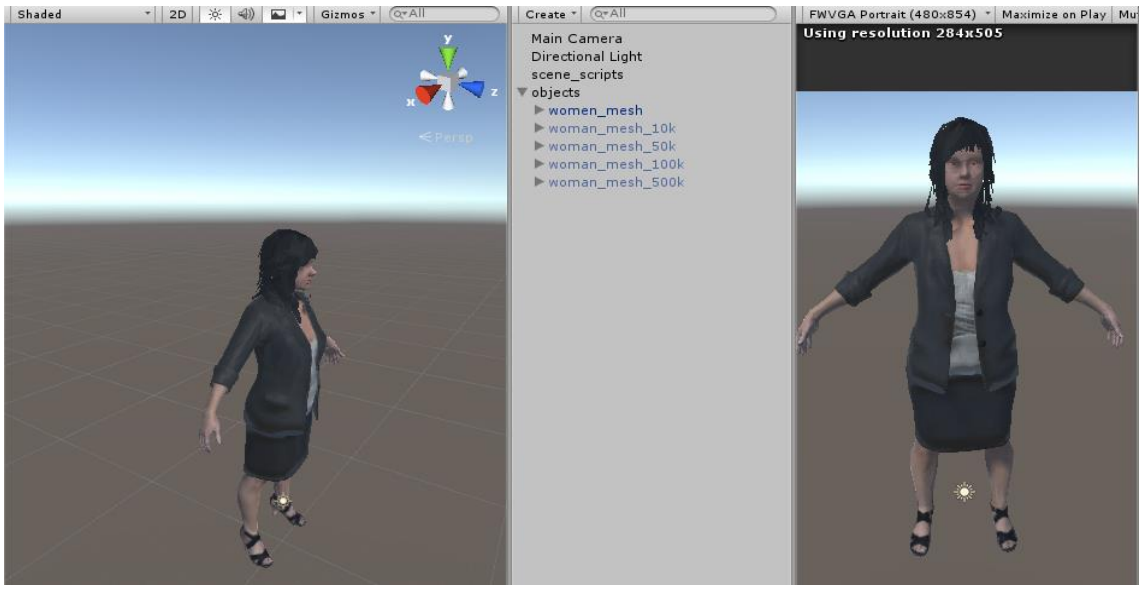

Abbildung 7: Szene 2 der Testanwendung mit einem Charaktermodell

Das verwendete 3D-Modell ist der Charakter *BWom0001_M3* aus den 3Ds Max Beispieldateien.[54]

Die 5 verwendeten Charaktermodelle werden nacheinander in verschiedenen Detailstufen eingeblendet. Die erste Detailstufe zeigt das Modell in der in den Beispieldateien vorliegenden Auflösung. Alle weiteren Modelle sind detailliertere Varianten des Modells. Der Detailgrad der Modelle steigt im Laufe der 25 Sekunden dauernden Szene von 6.384 auf 500.000 *Triangles*. Jeder der Charaktere verfügt über drei Materialien. Ein Material für die Kleidung und den Körper, eines für den Kopf und ein drittes für die Haare. Es werden jeweils die *PBR-Shader* verwendet.

[54] 3DS MAX 2015 SAMPLE FILES, http://knowledge.autodesk.com/support/3ds-max/downloads/caas/downloads/content/3ds-max-2015-sample-files.html, 05.10.15

2.3.4 Szene 3

Im letzten Abschnitt der Testanwendung kommt eine Szene aus der *PLAYMOBIL Scan* App zum Einsatz. Die Szene in Abbildung 8 zeigt 3 Fahrzeuge sowie 5 Charaktermodelle und ein Gebäude von *PLAYMOBIL*.

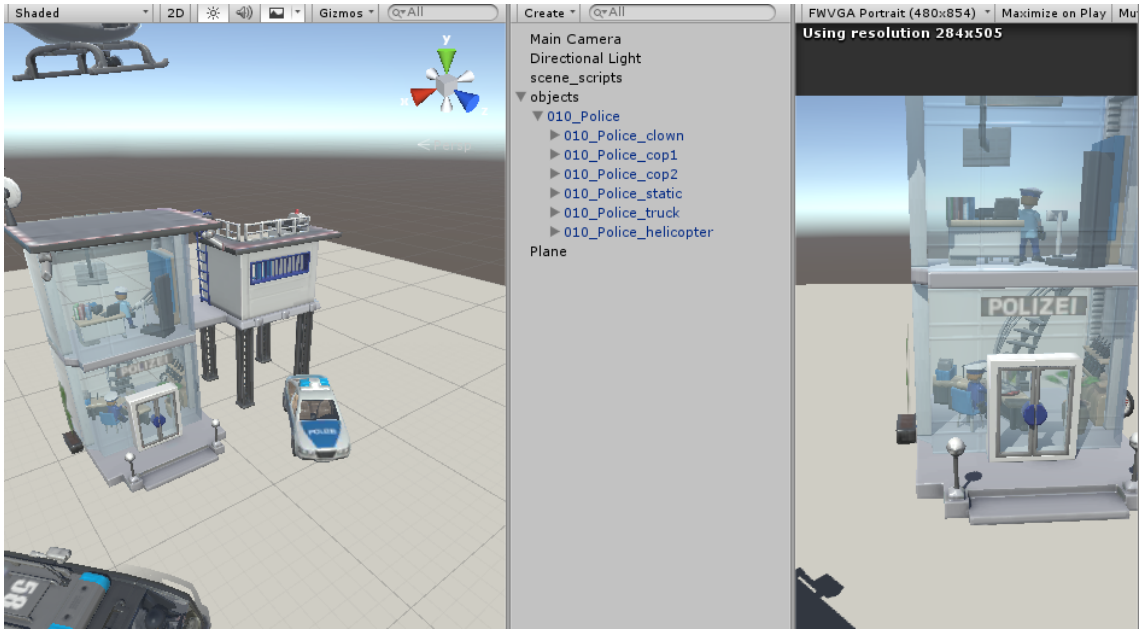

Abbildung 8: PLAYMOBIL-Modelle aus Szene 3 der Testanwendung

Die Beleuchtung entspricht der aus den Szenen 1 und 2. Um unterschiedlich hohe Auslastungen des Testgerätes zu erzeugen, wird die Kamera bewegt. Hierdurch schwankt die Zahl der 3D-Modelle im Kamerablickfeld. Dies hat zur Folge, dass zu verschiedenen Zeiten unterschiedlich hohe Zahlen von *Triangles* und geladenen *Texturen* im Bild zu sehen sind.

Wie bereits in Szene 1 und 2 werden auch hier die *PBR-Shader* eingesetzt.

Nach Ablauf von 25 Sekunden wird die Anwendung automatisch beendet. Dies geschieht über das *C# Skript Quit.cs*, das sich auf dem *scene_scripts*-Objekt befindet. Durch das Beenden der *Unity*-App werden auch keine *Profiler*-Daten mehr über *ADB* gesendet. Die Aufzeichnung ist zu diesem Zeitpunkt abgeschlossen und die Daten können ausgewertet werden.

2.4 Fazit

Für die Leistungsanalyse werden explizite Anforderungen an die Hard- und Software gestellt. Die Aufzeichnung der Leistungsdaten erfolgt über einen PC. Die Leistungsdaten des *Unity-Profilers* können nicht lokal auf dem *Android*-Gerät gespeichert werden. Nutzer die am Prozess der Leistungsanalyse mitwirken wollen, müssen den kompletten Versuchsaufbau reproduzieren. Dieser besteht aus einem *Android*-Gerät, welches via USB oder WLAN mit einem PC verbunden ist. Auf dem Computer muss das *Android-SDK* mit *ADB* und *DDMS* installiert sein. Das jeweils zu testende *Android*-Gerät muss außerdem im Entwicklermodus laufen. Denn nur so können Daten via *ADB* ausgegeben werden. Nur wenn diese Voraussetzungen erfüllt sind, kann eine Aufzeichnung der *Unity*-Leistungsdaten erfolgen. Eine Verbreitung der App, um Daten von möglichst vielen Geräten in kurzer Zeit zu sammeln, ist aufgrund der benötigten *ADB*-Verbindung und des benötigten Entwicklermodus nicht möglich. Jedes Gerät, das getestet werden soll, muss zwangsweise mit einem PC verbunden und in den Entwicklermodus versetzt werden.

3 Auswertung des Testszenarios

Im Folgenden Kapitel werden die im letzten Kapitel definierten Fragen zur Umsetzbarkeit einer Leistungsanalyse mobiler Endgeräte beantwortet.

3.1 Ist der Aufzeichnungsprozess reproduzierbar?

Der Aufzeichnungsprozess ist reproduzierbar, sobald bei mehreren Anwendungsdurchläufen annähernd gleiche Ergebnisse zustande kommen. Hierfür muss für jeden Durchlauf dieselbe Anzahl an *Profiler*-Datenblöcken vom *Android*-Gerät übertragen werden.

Um zu testen, ob alle Daten gesendet wurden, wurde ein *Skript* auf *Python*-Basis geschrieben. Das *Skript counter_v01.py* in Anlage 2 prüft die *Unity-Logfiles* auf alle *Profiler*-Datenblöcke. Da jeder Datenblock mit der Textzeile *Android Unity internal profiler stats* beginnt, wurde gezählt, wie oft diese Textzeile in einer *Logfile* vorkam. Das *Skript* durchläuft jede Zeile der *Unity Logfile*. Sobald die oben beschriebene Zeichenfolge registriert wurde, erhöht sich ein Zähler. Dieser Zähler hat bei der Initialisierung des *Skript*es den Wert null. Sobald das *Skript* alle Zeilen durchlaufen hat, gibt es die Anzahl *Profiler*-Datenblöcke aus.

Zunächst wurden 10 Tests mit der *Unity*-Testanwendung *3_scenes_125s.apk*[55] auf einem *Samsung Galaxy S3 GT-I9305* auf Testsystem 1 durchgeführt. Die Anwendungsdauer beträgt 125 Sekunden. Es müssen also 125 *Unity-Profiler*-Datenblöcke aufgezeichnet werden. Eine geringere Anzahl würde die Auswertbarkeit der Daten einschränken.

Zunächst wurden alle *Unity-Profiler*-Datensätze mit dem *counter_v01.py-Skript* auf ihre Vollständigkeit analysiert. Dabei stellte sich heraus, dass die Anzahl der *Profiler*-Blöcke nicht der Anwendungslänge von 125 Sekunden entsprach. Von den zu erwarteten 125 Datenblöcken wurden in jedem der 10 Durchläufe nur 86 aufgezeichnet.[56] Im folgenden Kapitel werden die aufgezeichneten Daten genauer analysiert.

[55] Anlage CD /apps/3_scenes_125s.apk

[56] Anlage CD /3scenes_125seconds/3scenes_125seconds_(01-10).txt

3.1.1 Datenanalyse

Für die genauere Auswertung der Datenblöcke wurde das *Python-Skript* erweitert. *Counter_v02.py* in Anlage 3 zählt neben der Gesamtzahl der *Profiler*-Blöcke auch die Anzahl der Blöcke für jeden Szenen- und Objektwechsel der App. Dies ermöglicht eine genauere Auswertung der aufgezeichneten Daten. Hierfür wurde das *Python-Skript* um eine weitere Abfrage erweitert. Diese prüft, ob die vordefinierten Textblöcke *switching object* und *changing scene* in der *Logfile* vorkommen. Beide Textblöcke werden über die Objekt- und Szenenwechsel-*Skripte* ausgegeben.

In Szene 1 wurden in den 10 Durchläufen zwischen 74 und 75 *Profiler*-Blöcke übertragen. Die Übertragung aller Blöcke erfolgte im Abstand von einer Sekunde. Für Szene 1 wurde in fast jedem Durchlauf die volle Anzahl an *Profiler*-Blöcken übertragen. Die folgende Liste zeigt einen *Logfile*-Auszug:

3scenes_125seconds_01.txt[57]

(...)
07-22 10:26:30.169: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:31.169: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:32.169: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:33.174: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:34.169: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:35.174: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:36.174: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:37.174: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:38.174: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:39.174: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:40.174: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:41.174: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:42.174: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:43.174: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:44.174: D/Unity(14094): Android Unity internal profiler stats:
(…)

[57] Anlage CD /3scenes_125seconds/3scenes_125seconds_01.txt

Der nachfolgende Abschnitt enthält alle Daten der 25 Sekunden langen zweiten Szene der Testanwendung. Von 25 zu erwartenden *Profiler*-Blöcken wurden lediglich 8 aufgezeichnet.

3scenes_125seconds_01.txt[58]

```
07-22 09:38:42.769: D/Unity(10872): Android Unity internal profiler stats:
07-22 09:38:44.694: D/Unity(10872): Android Unity internal profiler stats:
07-22 09:38:46.639: D/Unity(10872): Android Unity internal profiler stats:
07-22 09:38:48.669: D/Unity(10872): Android Unity internal profiler stats:
07-22 09:38:50.779: D/Unity(10872): Android Unity internal profiler stats:
07-22 09:38:53.054: D/Unity(10872): Android Unity internal profiler stats:
07-22 09:38:56.334: D/Unity(10872): Android Unity internal profiler stats:
07-22 09:39:00.249: D/Unity(10872): Android Unity internal profiler stats:
07-22 09:39:06.974: D/Unity(10872): Android Unity internal profiler stats:
```

Es ist deutlich zu erkennen, dass es Lücken in der Aufnahme gibt. Die Abstände zwischen den *Profiler*-Blöcken betragen bereits zu Beginn der Testszene zwei Sekunden. Gegen Ende des Tests steigen die Zeitabstände auf drei bis vier und schließlich sechs Sekunden.

Im 25 Sekunden dauernden dritten Testabschnitt wurden lediglich 4 von 25 Datenblöcken aufgezeichnet.

3scenes_125seconds_01.txt[59]

```
07-22 10:28:07.219: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:28:16.039: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:28:23.344: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:28:28.749: D/Unity(14094): Android Unity internal profiler stats:
```

[58] Anlage CD /3scenes_125seconds/3scenes_125seconds_01.txt

[59] Ebenda

In einem weiteren Test wurde überprüft, ob die Anzahl der Blöcke zwischen dem Objektwechsel variiert. Für diesen Test wird nur der zweite Szenenabschnitt mit den sich verändernden Polygonzahlen analysiert.

3scenes_125seconds_01.txt[60]

07-22 10:27:36.199: D/Unity(14094): Android Unity internal profiler stats:
objectchange - processed datablocks: 75
07-22 10:27:40.754: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:27:42.754: D/Unity(14094): Android Unity internal profiler stats:
objectchange - processed datablocks: 77
07-22 10:27:44.894: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:27:47.144: D/Unity(14094): Android Unity internal profiler stats:
objectchange - processed datablocks: 79
07-22 10:27:49.774: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:27:53.189: D/Unity(14094): Android Unity internal profiler stats:
objectchange - processed datablocks: 81
07-22 10:27:57.499: D/Unity(14094): Android Unity internal profiler stats:
objectchange - processed datablocks: 82

3scenes_125seconds_02.txt[61]

07-22 10:23:30.684: D/Unity(13718): Android Unity internal profiler stats:
07-22 10:23:32.574: D/Unity(13718): Android Unity internal profiler stats:
07-22 10:23:34.514: D/Unity(13718): Android Unity internal profiler stats:
objectchange - processed datablocks: 77
07-22 10:23:36.519: D/Unity(13718): Android Unity internal profiler stats:
07-22 10:23:38.589: D/Unity(13718): Android Unity internal profiler stats:
objectchange - processed datablocks: 79
07-22 10:23:40.759: D/Unity(13718): Android Unity internal profiler stats:
07-22 10:23:43.904: D/Unity(13718): Android Unity internal profiler stats:
objectchange - processed datablocks: 81
07-22 10:23:47.514: D/Unity(13718): Android Unity internal profiler stats:
objectchange - processed datablocks: 82
07-22 10:23:53.009: D/Unity(13718): Android Unity internal profiler stats:
scenechange - processed datablocks: 83

[60] Anlage CD /3scenes_125seconds/3scenes_125seconds_01.txt

[61] Anlage CD /3scenes_125seconds/3scenes_125seconds_02.txt

3scenes_125seconds_05.txt[62]

```
07-22 10:10:22.059: D/Unity(12657): Android Unity internal profiler stats:
07-22 10:10:23.944: D/Unity(12657): Android Unity internal profiler stats:
objectchange - processed datablocks:  77
07-22 10:10:25.954: D/Unity(12657): Android Unity internal profiler stats:
07-22 10:10:28.179: D/Unity(12657): Android Unity internal profiler stats:
objectchange - processed datablocks:  79
07-22 10:10:30.554: D/Unity(12657): Android Unity internal profiler stats:
07-22 10:10:34.039: D/Unity(12657): Android Unity internal profiler stats:
objectchange - processed datablocks:  81
07-22 10:10:38.209: D/Unity(12657): Android Unity internal profiler stats:
objectchange - processed datablocks:  82
```

Diese drei Auszüge aus unterschiedlichen *Logfiles* derselben Szene zeigen deutlich die schwankende Anzahl der *Profiler*-Blöcke zwischen dem Objektwechsel. Der Aufzeichnungsprozess ist unter diesen Bedingungen nicht reproduzierbar.

In einem weiteren Test wurde geprüft, ob die Lücken in den *Logfiles* durch die Verwendung mehrerer Szenen auftreten. Hierfür wurde die Szene 3 allein ausgeführt. Die Dauer des Tests wurde auf 300 Sekunden gesetzt. Neben den Lücken in den *Logfiles* wurde mit *1_scene_300s.apk*[63] überprüft, ob die Anzahl der Datenblöcke bei längeren Szenen einer Schwankung unterliegt. Im folgenden Abschnitt sind die ermittelten Blockzahlen der *Unity-Logfiles* aufgeführt.

1scene_300seconds_(01-10).txt[64]

```
1scene_300seconds_01.txt        processed datablocks:  61
1scene_300seconds_02.txt        processed datablocks:  62
1scene_300seconds_03.txt        processed datablocks:  62
1scene_300seconds_04.txt        processed datablocks:  60
1scene_300seconds_05.txt        processed datablocks:  61
1scene_300seconds_06.txt        processed datablocks:  60
1scene_300seconds_07.txt        processed datablocks:  63
1scene_300seconds_08.txt        processed datablocks:  60
1scene_300seconds_09.txt        processed datablocks:  62
1scene_300seconds_10.txt        processed datablocks:  63
```

[62] Anlage CD /3scenes_125seconds/3scenes_125seconds_05.txt

[63] Anlage CD /apps/1_scene_300s.apk

[64] Anlage CD /1scene_300seconds/1scene_300seconds_(01-10).txt

Anhand der 10 aufgenommenen *Logfiles* lässt sich die durchschnittliche Anzahl der aufgezeichneten *Profiler*-Blöcke sowie die Standardabweichung bestimmen.

$$\emptyset_n = \frac{\Sigma n}{n}$$

$$\emptyset = 61{,}27$$

$$\sigma_n = \sqrt{\frac{(n1 - \emptyset)^2 + (n2 - \emptyset)^2 + (\dots)}{n}}$$

$$\sigma_n = \sqrt{\frac{(62 - 61{,}27)^2 + (62 - 61{,}27)^2 + (\dots)}{10}}$$

$$\sigma = 1{,}19$$

Im Durchschnitt wurden 61,27 *Profiler*-Blöcke pro Durchgang aufgezeichnet, dies resultiert in einem Datenverlust von 79,57 %. Die Standardabweichung beträgt 1,19 *Profiler*-Blöcke.

Um zu ermitteln, ob die Abweichung bei längeren Szenen steigt, wurde die Szenenzeit verdoppelt. Hierbei wurde die App mit der Bezeichnung *1_scene_600s.apk*[65] verwendet. Bei 10 Durchläufen einer 600 Sekunden langen Szene kamen folgende Ergebnisse zustande.

1scene_300seconds_(01-10).txt[66]

1scene_600seconds_01.txt	processed datablocks: 132
1scene_600seconds_02.txt	processed datablocks: 132
1scene_600seconds_03.txt	processed datablocks: 132
1scene_600seconds_04.txt	processed datablocks: 132
1scene_600seconds_05.txt	processed datablocks: 132
1scene_600seconds_06.txt	processed datablocks: 132
1scene_600seconds_07.txt	processed datablocks: 132
1scene_600seconds_08.txt	processed datablocks: 131
1scene_600seconds_09.txt	processed datablocks: 131
1scene_600seconds_10.txt	processed datablocks: 131

[65] Anlage CD /apps/1_scene_600s.apk

[66] Anlage CD /1scene_600seconds/1scene_600seconds_(01-10).txt

Die Standardabweichung ist durch die Erhöhung der Szenendauer auf 0,48 Datenblöcke gesunken. Die Anzahl der *Profiler*-Blöcke schwankte bei 10 Tests zwischen 131 und 132. Daraus resultiert, dass ein einzelner langer Test geringere Abweichungen hervorbringt. Jedoch fehlen zwischen 468 und 469 der *Profiler*-Blöcke. Dies entspricht einem Datenverlust von 78 %. Der Datenverlust ist bei einer 600 Sekunden langen Szene nur minimal geringer als bei einer Länge von 300 Sekunden.

Anhand der analysierten Daten wird deutlich, dass bei komplexen Szenen Verluste bei der Übertragung der *Profiler*-Blöcke auftreten. Die Datenverluste bei komplexen Szenen fallen sehr hoch aus. Im folgenden Kapitel wird überprüft wie die Datenverluste Zustandekommen.

3.1.2 Fehleranalyse

Um zu überprüfen, wodurch die fehlenden *Profiler*-Blöcke hervorgerufen wurden, mussten die *Logfiles* auf ihre Inhalte analysiert werden. Die größten Verluste von *Profiler*-Blöcken traten in den komplexen Szenen 2 und 3 auf. In diesen Szenen war die *Frame Rate* der App sehr gering.

In diesem Abschnitt wird analysiert, ob es einen direkten Zusammenhang zwischen der *Frame Rate* und den fehlenden *Profiler*-Blöcken gibt. In einer *Unity-Profiler*-Datei ist die *Frame Rate* nicht direkt angegeben. Stattdessen werden sie als *Frame Time* dargestellt. Dies beschreibt die Zeit in Millisekunden die für die Berechnung eines *Frames* notwendig ist. Die *Frame Rate* einer Anwendung, die in *fps* angegeben wird, lässt sich anhand folgender Formel aus der *Frame Time* ableiten.

$$FPS = \frac{1000\ ms}{t_{Frame}}$$

In Szene 1 betrug die *Frame Time* in den meisten *Profiler*-Blöcken 16,7 ms. Daraus ergibt sich folgende Formel für die *Frame Rate*.

$$FPS = \frac{1000\ ms}{16{,}7\ ms}$$

$$FPS = 59{,}88\ \text{fps}$$

Daraus resultiert, dass bei einer *Frame Time* von 16,7 ms eine *Frame Rate* von 60 fps erreicht wird. Ein höherer Wert ist unter *Android* nicht möglich. Ab *Android 4.1 Jelly Bean* liegt die Bildwiederholrate des Betriebssystems bei 60 Hz, das heißt, es können maximal 60 fps angezeigt werden.[67] Niedrigere Werte der als 16,7 ms und demzufolge mehr als 60 fps sind aus diesem Grund nicht möglich.

Der folgende Auszug stellt alle *Frame Time* Abschnitte der *Profiler*-Blöcke des ersten Testlaufs dar.

3scenes_125seconds_01.txt[68]

```
07-22 10:26:22.169: D/Unity(14094): frametime>    min:  0.0   max: 28.0   avg: 16.3
07-22 10:26:23.169: D/Unity(14094): frametime>    min: 16.0   max: 17.3   avg: 16.7
07-22 10:26:24.169: D/Unity(14094): frametime>    min: 15.9   max: 17.4   avg: 16.7
(…)
07-22 10:27:32.184: D/Unity(14094): frametime>    min: 16.2   max: 17.1   avg: 16.7
07-22 10:27:33.184: D/Unity(14094): frametime>    min: 15.8   max: 17.5   avg: 16.7
07-22 10:27:34.199: D/Unity(14094): frametime>    min: 15.8   max: 33.4   avg: 16.9
07-22 10:27:35.199: D/Unity(14094): frametime>    min: 16.4   max: 16.9   avg: 16.7
scenechange - processed datablocks:  74
07-22 10:27:36.199: D/Unity(14094): frametime>    min: 16.4   max: 16.9   avg: 16.7
07-22 10:27:40.754: D/Unity(14094): frametime>    min:  7.4   max: 2663.6   avg: 75.6
07-22 10:27:42.759: D/Unity(14094): frametime>    min: 29.0   max: 37.1   avg: 33.3
07-22 10:27:44.894: D/Unity(14094): frametime>    min: 25.9   max: 43.7   avg: 35.6
07-22 10:27:47.144: D/Unity(14094): frametime>    min: 32.8   max: 42.7   avg: 37.6
07-22 10:27:49.774: D/Unity(14094): frametime>    min: 28.1   max: 74.4   avg: 43.5
07-22 10:27:53.189: D/Unity(14094): frametime>    min: 50.7   max: 64.3   avg: 56.9
07-22 10:27:57.499: D/Unity(14094): frametime>    min: 41.2   max: 95.5   avg: 71.6
scenechange - processed datablocks:  82
07-22 10:28:07.219: D/Unity(14094): frametime>    min: 15.8   max: 2083.1   avg: 160.6
07-22 10:28:16.039: D/Unity(14094): frametime>    min: 110.5   max: 271.9   avg: 147.0
07-22 10:28:23.344: D/Unity(14094): frametime>    min: 95.7   max: 182.2   avg: 122.5
07-22 10:28:28.749: D/Unity(14094): frametime>    min: 70.0   max: 110.1   avg: 90.5
```

In Szene 1 lief die Anwendung mit einer durchschnittlichen *Frame Time* von 16,7 ms. Die *Profiler*-Blöcke werden in diesem Abschnitt mit einem Abstand von einer Sekunde übertragen.

Ab Szene 2 steigt die *Frame Time* stark an. Parallel dazu erhöht sich auch der Abstand zwischen den *Profiler*-Blöcken. Eine annähernde Verdopplung der *Frame Time* auf etwa 33,7 ms resultiert darin, dass die *Profiler*-Blöcke nur im Abstand von 2 Sekunden übertragen werden. Dies ist im folgenden *Logfile*-Auszug deutlich zu erkennen.

---

[67] Vgl. IHLENFELD,2012

[68] Anlage CD /3scenes_125seconds/3scenes_125seconds_01.txt

3scenes_125seconds_02.txt[69]

```
07-22 10:23:36.519: D/Unity(13718): frametime>    min: 28.5   max: 41.5   avg: 33.4
07-22 10:23:38.594: D/Unity(13718): frametime>    min: 32.1   max: 36.9   avg: 34.6
```

Höhere Werte der *Frame Time* resultieren in deutlich höheren Zeitabständen der *Profiler*-Blöcke. Bei einer durchschnittlichen *Frame Time* von 64,9 ms beträgt der Abstand zum nächsten Block bereits 4 Sekunden.

3scenes_125seconds_10.txt[70]

```
07-22 09:38:56.334: D/Unity(10872): frametime>    min: 48.7   max: 62.8   avg: 54.7
07-22 09:39:00.249: D/Unity(10872): frametime>    min: 43.6   max: 97.9   avg: 64.9
```

Daraus ergibt sich folgendes: Der Abstand der *Profiler*-Blöcke steht in direktem Zusammenhang zwischen der *Frame Time* und der Zeit die das *Android*-System für die Einzelbildberechnung benötigt.

Anhand der *Frame Time* des *Android*-Systems und der aktuellen *Frame Time* lässt sich ein Faktor zu bestimmen welcher den Abstand der *Profiler*-Blöcke in Sekunden erhöht. Eine aktuelle *Frame Time* von 16,7ms ergibt bei einer *Frame Time* des Systems von 16,7ms einen Abstand von exakt einer Sekunde. Erhöht sich die *Frame Time*, so erhöht sich der zeitliche Abstand. Die folgende Formel gilt für den Fall, dass die Systembildrate 60 Hz beträgt. Die Bildberechnungszeit t_{Frame} muss in Millisekunden angegeben werden.

$$t_{Block} = \frac{t_{Frame}}{16{,}7\ ms} \times 1\ s$$

[69] Anlage CD /3scenes_125seconds/3scenes_125seconds_02.txt
[70] Anlage CD /3scenes_125seconds/3scenes_125seconds_10.txt

Für die beiden zuvor aufgeführten *Logfile*-Ausschnitte ergeben sich folgende Berechnungen.

3scenes_125seconds_02.txt[71]

$$t_{Block} = \frac{34{,}4}{16{,}7} \times 1\ s$$

$$t_{Block} = 2{,}06\ s$$

3scenes_125seconds_10.txt[72]

$$t_{Block} = \frac{64{,}9}{16{,}7} \times 1\ s$$

$$t_{Block} = 3{,}89\ s$$

Die Abweichungen zwischen der berechneten Zeit und den eigentlichen Zeitabständen sind ein Resultat der verwendeten durchschnittlichen *Frame Time.* Zudem wird in den *Logfiles* nur eine Nachkommastelle angegeben.

Aus der Fehleranalyse folgt, dass die fehlenden *Profiler*-Blöcke das Resultat einer zu hohen *Frame Time* sind. Höhere Werte als 16,7 ms resultieren in einem höheren Abstand zwischen den *Profiler*-Blöcken. Niedrigere Werte als 16,7 ms sind aufgrund der Bildratenlimitierung nicht möglich. Um eine fehlerfreie Ausgabe aller *Profiler*-Blöcke zu gewährleisten, müssen also 60 *fps* berechnet werden. Im folgenden Kapitel wird darauf eingegangen, wie das *Rendern* von 60 *fps* erzwungen werden kann um alle *Profiler*-Blöcke zu übertragen.

[71] Anlage CD /3scenes_125seconds/3scenes_125seconds_02.txt
[72] Anlage CD /3scenes_125seconds/3scenes_125seconds_10.txt

3.1.3 Fehlerbehebung

Um die *Profiler*-Lücken zu schließen, wird das *Rendern* von 60 fps erzwungen. Dadurch läuft die Anwendung, wenn die Bildrate unter 60 *fps* sinkt, in geringerer Geschwindigkeit ab. Der Wert von 60 fps wurde als Ausgangswert gewählt, da die Bildwiederholrate von *Android*-Geräten bei 60 Hz liegt. Um die *60* fps zu erzwingen wurde das *Skript Profiler_Framerate.cs* verwendet. Der *C#*-Code in Anlage 7 ist eine Abwandlung eines *Skripts* aus der *Unity*-Dokumentation.[73] Das ursprüngliche *Skript* erzwingt das *Rendern* aller *Frames*, um diese auf der Festplatte speichern zu können.

Zunächst musste überprüft werden, ob der *Unity*-Zeitgeber, durch die geringere Geschwindigkeit der Anwendung bei hoher Last, negativ beeinflusst wird. Eine Verlangsamung des Zeitgebers hätte fehlerhafte Objekt- und Szenenwechsel zur Folge. Hierfür wurden 10 Testszenen aufgezeichnet, welche dem ersten Testdurchlauf entsprechen.[74]

3scenes_125seconds_60fps_01.txt[75]

```
07-23 13:13:52.473: D/Unity(2790): Android Unity internal profiler stats:
(…)
07-23 13:14:06.473: D/Unity(2790): Android Unity internal profiler stats:
objectchange - processed datablocks:  15
07-23 13:14:07.473: D/Unity(2790): Android Unity internal profiler stats:
(…)
07-23 13:14:21.483: D/Unity(2790): Android Unity internal profiler stats:
objectchange - processed datablocks:  30
07-23 13:14:22.483: D/Unity(2790): Android Unity internal profiler stats:
(…)
07-23 13:15:29.078: D/Unity(2790): Android Unity internal profiler stats:
objectchange - processed datablocks:  85
07-23 13:15:32.203: D/Unity(2790): Android Unity internal profiler stats:
(…)
07-23 13:15:45.308: D/Unity(2790): Android Unity internal profiler stats:
objectchange - processed datablocks:  90
07-23 13:15:49.423: D/Unity(2790): Android Unity internal profiler stats:
(…)
07-23 13:16:06.728: D/Unity(2790): Android Unity internal profiler stats:
objectchange - processed datablocks:  95
07-23 13:16:15.778: D/Unity(2790): Android Unity internal profiler stats:
(…)
processed datablocks:  125
```

In allen 10 Durchläufen wurden alle 125 Datenblöcke ausgegeben. Die Datenblöcke der Objektwechsel sind ebenfalls korrekt. Alle 10 Durchläufe kamen zu demselben Ergebnis.

---

[73] Vgl. UNITY, http://docs.unity3d.com/ScriptReference/Time-captureFramerate.html, 31.08.15

[74] Anlage CD /3scenes_125seconds_60fps/System_1/3scenes_125seconds_60fps_(01-10).txt

[75] Anlage CD /3scenes_125seconds_60fps/System_1/3scenes_125seconds_60fps_01.txt

Daraus folgt, dass der Zeitgeber durch die erzwungene Bildrate ebenfalls verlangsamt wird. In der ersten Szene werden 15 Datenblöcke pro Objekt ausgegeben. In der zweiten Szene werden 5 Blöcke gesendet. Die dritte Szene enthält keine Objektwechsel, dementsprechend gibt es nur die Gesamtzahl der Datenblöcke.

Die Zeit zur Erstellung eines *Logfiles* hat sich jedoch signifikant erhöht. Der erste Testdurchlauf mit einer Länge von 125 Sekunden lief in Echtzeit ab. Bei dem zweiten Testdurchlauf mit den erzwungenen 60 *fps* beträgt die Gesamtdauer des dritten Szenenabschnitts 2 Minuten 40 Sekunden. Die Gesamtdauer der Testanwendung mit 60 fps betrug 6 Minuten 9 Sekunden. Daraus resultiert einer Steigerung der Aufzeichnungszeit um 4 Minuten 4 Sekunden.

In einer weiteren Testreihe wurde überprüft, wie sich eine Verringerung der erzwungenen Bildrate auswirkt. Eine niedrigere erzwungene *Frame Rate* hätte eine Verkürzung der Testdauer bei komplexen Szenen zur Folge. Ein erster Test wurde mit einer Bildrate von 30 fps durchgeführt.

3scenes_125seconds_30fps_01.txt[76]

```
07-23 15:56:12.870: D/Unity(28292): Android Unity internal profiler stats:
(…)
07-23 15:56:18.870: D/Unity(28292): Android Unity internal profiler stats:
objectchange - processed datablocks:  7
07-23 15:56:19.870: D/Unity(28292): Android Unity internal profiler stats:
(…)
07-23 15:56:26.870: D/Unity(28292): Android Unity internal profiler stats:
objectchange - processed datablocks:  15
07-23 15:56:27.870: D/Unity(28292): Android Unity internal profiler stats:
(…)
07-23 15:56:33.870: D/Unity(28292): Android Unity internal profiler stats:
objectchange - processed datablocks:  22
07-23 15:56:34.870: D/Unity(28292): Android Unity internal profiler stats:
(…)
07-23 15:59:13.145: D/Unity(28292): Android Unity internal profiler stats:
processed datablocks:  62
```

Durch die Reduktion auf 30 *fps* reduzierten sich auch die gesendeten *Profiler*-Blöcke um die Hälfte. In jedem der 10 Durchläufe wurden 62 Blöcke übertragen. Dies entspricht einer Halbierung der Anzahl der übertragenen Blöcke. Die Zahl 62 kommt zustande, da es nicht möglich ist halbe *Frames* zu übertragen. Die Dauer des Tests betrug 3 Minuten und 5 Sekunden.

[76] Anlage CD /3scenes_125seconds_30fps/3scenes_125seconds_30fps_01.txt

Es wurden zwei weitere Testdurchläufe mit 45 und 120 fps durchgeführt. Eine *Frame Rate* von 120 resultierte in einer Verdopplung der Datensätze. So das schließlich 2 *Profiler*-Blöcke pro Sekunde übertragen wurden. Die für den Test benötigte Zeit erhöhte sich auf 8 Minuten und 4 Sekunden.

Bei 45 *fps* wurden von 93 von 125 Datensätzen übertragen. Diese Zahlen waren in allen 10 Durchläufen konstant.[77]

3scenes_125seconds_45fps_01.txt[78]

```
07-24 17:05:26.795: D/Unity(8855): Android Unity internal profiler stats:
(...)
07-24 17:05:36.795: D/Unity(8855): Android Unity internal profiler stats:
objectchange - processed datablocks:  11
07-24 17:05:37.795: D/Unity(8855): Android Unity internal profiler stats:
(...)
07-24 17:05:47.795: D/Unity(8855): Android Unity internal profiler stats:
objectchange - processed datablocks:  22
07-24 17:05:48.810: D/Unity(8855): Android Unity internal profiler stats:
(...)
07-24 17:09:54.020: D/Unity(8855): Android Unity internal profiler stats:
processed datablocks:  93
```

Ein Abweichen von der *Frame Rate* des Betriebssystems resultiert anhand der durchgeführten Tests in einer höheren oder niedrigeren Zahl an Datensätzen. Bei sehr schnellen 3D-Szenen, in denen Inhalte mit hoher Frequenz wechseln, kann eine höhere *Frame Rate* von Nutzen sein um kleinere Zeitabstände erfassen zu können. Anwendungsfälle für eine höhere Datenrate wären Partikeleffekte wie Explosionen. Um eine Verdopplung oder Verdreifachung der Datensätze zu erreichen, muss darauf geachtet werden ein Vielfaches der Systembildrate zu verwenden. Bei den 60 fps bei *Android* wären dies entsprechend 120, 180 oder 240 fps. Zudem erhöht sich hierdurch die mittels *ADB* übertragene Datenmenge signifikant. In Kapitel 6.1 wird näher auf diesen Sachverhalt eingegangen. Die Verwendung einer niedrigeren *Frame Rate* resultiert in lückenhaften Datensätzen. Objektwechsel oder Effekte, die in Abständen von einer Sekunde ablaufen, können hierdurch nicht mehr erfasst werden. Alle weiteren Tests wurden deshalb mit 60 fps durchgeführt. Höhere Werte als 60 fps wurden nicht genutzt, da sich die Testdauer und die Menge der aufgezeichneten Daten stark erhöhte.

[77] Anlage CD /3scenes_125seconds_45fps/3scenes_125seconds_45fps_(01-10).txt

[78] Anlage CD /3scenes_125seconds_30fps/3scenes_125seconds_45fps_01.txt

3.2 Beeinflusst die Länge der Testanwendung die aufgenommenen Daten

Seitens des *Android*-Gerätes gibt es keine Beeinflussung durch die Länge der aufgenommenen Daten. Denn die Daten werden in Echtzeit vom Gerät, über *ADB,* an den PC übertragen und mit *DDMS* aufgezeichnet. In diesem Kapitel wird geprüft, ob die Daten am Aufzeichnungssystem verfälscht werden können, wenn die Testanwendung zu lang ist.

Tests mit einer mehrere Minuten dauernden Testszene führten am PC zu einer Überladung des *Buffers* in *DDMS*. Dies resultierte darin, dass für jeden neu empfangenen Datensatz ein älterer gelöscht wurde. Standardmäßig werden nur die letzten 5000 Zeilen zwischengespeichert. Bei einer mehrere Minuten laufenden Anwendung können diese schnell erreicht werden. Denn neben den *Unity*-Meldungen müssen auch alle *Android* Meldungen zwischengespeichert werden. Im folgenden Absatz sind zwei *Android*-Systemmeldungen zwischen drei *Unity-Profiler*-Blöcken zu sehen. Meldungen des *Android*-Systems wurden fett markiert.

3scenes_125seconds_60fps_no_filter.txt[79]

07-29 15:58:51.844: D/Unity(18514): mono-scripts> update: 0.1 (…)
07-29 15:58:51.844: D/Unity(18514): mono-memory> used heap: 311296 (…)
07-29 15:58:51.844: D/Unity(18514): ----------
07-29 15:58:52.619: D/STATUSBAR-NetworkController(2813): (…)
07-29 15:58:52.624: D/STATUSBAR-NetworkController(2813): Nothing, (…)
07-29 15:58:54.144: D/SensorService(2410): [SO] 0.316 -0.134 9.548
07-29 15:58:54.944: D/Unity(18514): Android Unity internal profiler stats:
07-29 15:58:54.944: D/Unity(18514): cpu-player> min: 41.7 max: 64.3 (…)

Ein *Unity-Profiler*-Block benötigt 15 Zeilen. Da zusätzlich zu den *Profiler*-Daten weitere *Unity* und *Android*-Daten gesendet werden, wird der Standardwert von 5000 schnell erreicht. Bei der Aufnahme von 3 Szenen mit einer Gesamtlänge von 125 Sekunden bei 60 fps wurden etwa 4250 Zeilen empfangen. Das Gerät wurde bei der Aufnahme nicht berührt und es gab keine eingehenden Nachrichten. Eingaben über den Touchscreen, eingehende Nachrichten oder das Auslösen der Bewegungssensoren würden ebenfalls *Debug*-Daten über *ADB* senden. Hierdurch kann der *Buffer* noch schneller gefüllt wer-

[79] Anlage CD /3scenes_125seconds_60fps/System_1/3scenes_125seconds_60fps_no_filter.txt

den. Dieses Problem konnte durch vergrößern des *Buffers* in DDMS gelöst werden. Abbildung 9 zeigt die angepasste *Buffer*-Einstellung um mehr *ADB*-Meldungen zu speichern.

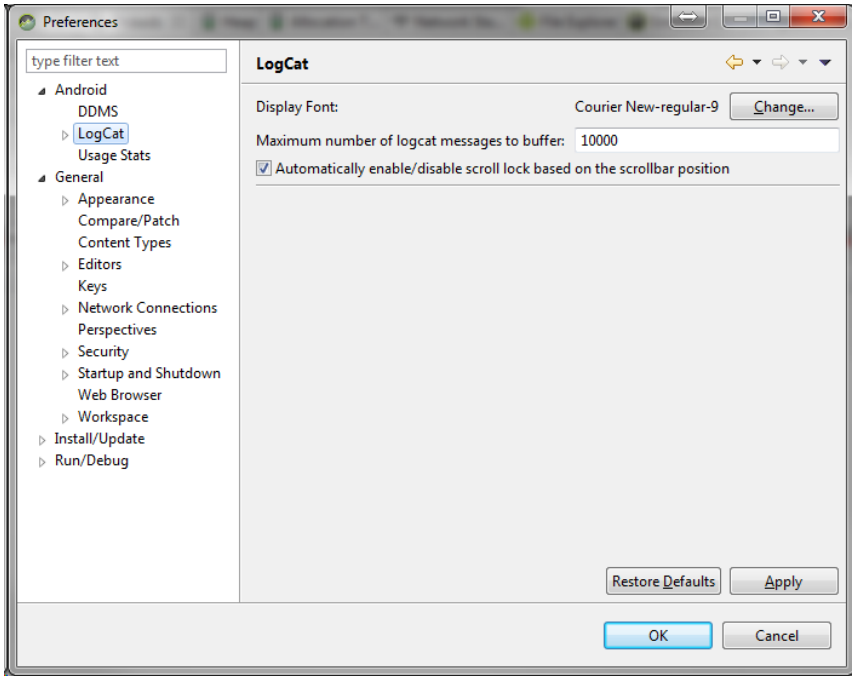

Abbildung 9: Einstellung des Buffers für ADB

3.3 Sind die Testdaten statistisch auswertbar?

Die über *ADB* empfangenen Daten werden in Blöcken geschrieben. Diese Daten sind für die statistische Auswertung nicht geeignet. Um die Daten verwerten zu können, müssen diese in eine zeilenweise lesbare Form gebracht werden. Jede Zeile stellt hierbei einen Zeitabschnitt mit allen Parametern dar. Im folgenden Absatz ist ein *Profiler*-Datenblock dargestellt wie er mit *DDMS* aufgezeichnet wird.

3scenes_125seconds_60fps_01.txt[80]

```
07-22 10:26:22.169: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:22.169: D/Unity(14094): cpu-player>    min:  0.0   max: 27.9  (…)
07-22 10:26:22.169: D/Unity(14094): cpu-ogles-drv> min:  0.0   max:  0.0   (…)
07-22 10:26:22.169: D/Unity(14094): cpu-present>   min:  0.0   max:  0.9   (…)
07-22 10:26:22.169: D/Unity(14094): frametime>     min:  0.0   max: 28.0   (…)
07-22 10:26:22.169: D/Unity(14094): batches>       min:  0   max:  4   avg:  3
07-22 10:26:22.169: D/Unity(14094): draw calls>    min:  0   max:  4   avg:  3
07-22 10:26:22.169: D/Unity(14094): tris>          min:    0  max:  3218  avg:  3164
07-22 10:26:22.169: D/Unity(14094): verts>         min:    0  max:  6074  avg:  5972
07-22 10:26:22.169: D/Unity(14094): dynamic batching> batched draw calls: (…)
07-22 10:26:22.169: D/Unity(14094): static batching>  batched draw calls: (…)
07-22 10:26:22.169: D/Unity(14094): player-detail> physx:  0.6 animation: (…)
07-22 10:26:22.169: D/Unity(14094): mono-scripts>  update:  0.0    (…)
07-22 10:26:22.169: D/Unity(14094): mono-memory>   used heap: 31948 (…)
```

Für die statistische Auswertung müssen die Daten in die in Tabelle 2 gezeigte Form gebracht werden. Eine Zeile der Tabelle entspricht einem *Profiler*-Block. Im folgenden Testabschnitt wird mit bereits umgewandelten *Logfiles* gearbeitet. Die Umwandlung der Daten wird im Kapitel 3.6 genauer beschrieben.

	Frame Time			*Draw Calls*		(…)
Data	*min*	*max*	*avg*	*min*	*max*	(…)
1	0	28,5	16,3	0	4	(…)
(…)	(…)	(…)	(…)	(…)	(…)	(…)

Tabelle 2: Konvertierte Logfile

[80] Anlage CD /3scenes_125seconds_60fps/System_1/3scenes_125seconds_60fps_01.txt

Für den Test wurde die Anwendung *3scenes_125seconds_60.apk*[81] genutzt. Zur Aufzeichnung der Daten wurde *System 1* verwendet. Zunächst wurde anhand der Testreihe *3scenes_125seconds_60fps_(01-10).txt*[82] geprüft, ob die Daten reproduzierbar sind. Hierfür wurden Standardabweichungen für verschiedene Parameter berechnet.

Die Standardabweichung der *Frame Time* des ersten Testabschnitts liegt im Durchschnitt bei 0,01 ms. Die minimale Abweichung beträgt 0,0 ms. Bei der maximalen Standardabweichung werden 0,063 ms erreicht. Im zweiten Testabschnitt lag die durchschnittliche Standardabweichung bei 1,185 ms. Die Werte variieren zwischen 0,399 ms und 2,425 ms. Der Dritte Testblock weißt mit 3,476 ms die höchste durchschnittliche Standardabweichung pro Datenblock auf. Die Messwerte weichen zwischen 1,181 ms und 6,964 ms voneinander ab. Die Testabschnitte 2 und 3 weisen eine höhere Komplexität auf als der erste Abschnitt. Die Standardabweichung der Testergebnisse steigt mit einer höheren Szenenkomplexität. Abbildung 10 zeigt die Verläufe der *Frame Time*-Werte aller zehn Testdurchläufe.

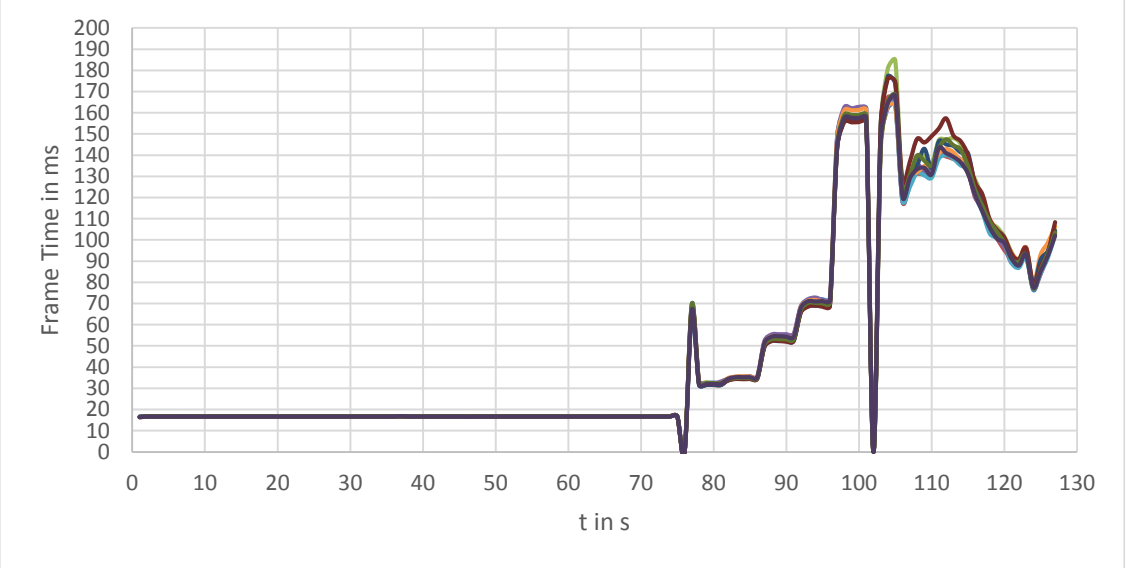

Abbildung 10: Frame Time Verlauf

Es ist deutlich zu erkennen, dass bei steigender Szenenkomplexität höhere Abweichungen möglich sind. Bis Sekunde 95 liegen die Kurven dicht beieinander. In den Sekunden 90 bis 95 wird ein Charaktermodell mit 100.000 *Triangles* angezeigt. Nach dem Wechsel auf ein Modell mit 500.000 *Triangles* steigt die Abweichung zwischen den einzelnen

[81] Anlage CD /apps/3scenes_125seconds_60fps.apk

[82] Anlage CD /3scenes_125seconds_60fps/System_1/3scenes_125seconds_60fps_(01-10).txt

Durchläufen stark an. Die durchschnittliche Standardabweichung vom Mittelwert beträgt in den von 90 bis 95 Sekunden 1,234 ms. Bei den darauffolgenden 5 Sekunden, mit dem aus 500.000 *Triangles* bestehenden Charaktermodell, beträgt die durchschnittliche Abweichung zwischen den einzelnen *Logfiles* 2,349 ms.

Abbildung 11 zeigt den Verlauf der *CPU-Player*-Zeit. Der Verlauf der zehn Kurven ist ähnlich wie der *Frame Time*-Verlauf. Die Ungenauigkeiten steigen auch hier mit höherer Szenenkomplexität.

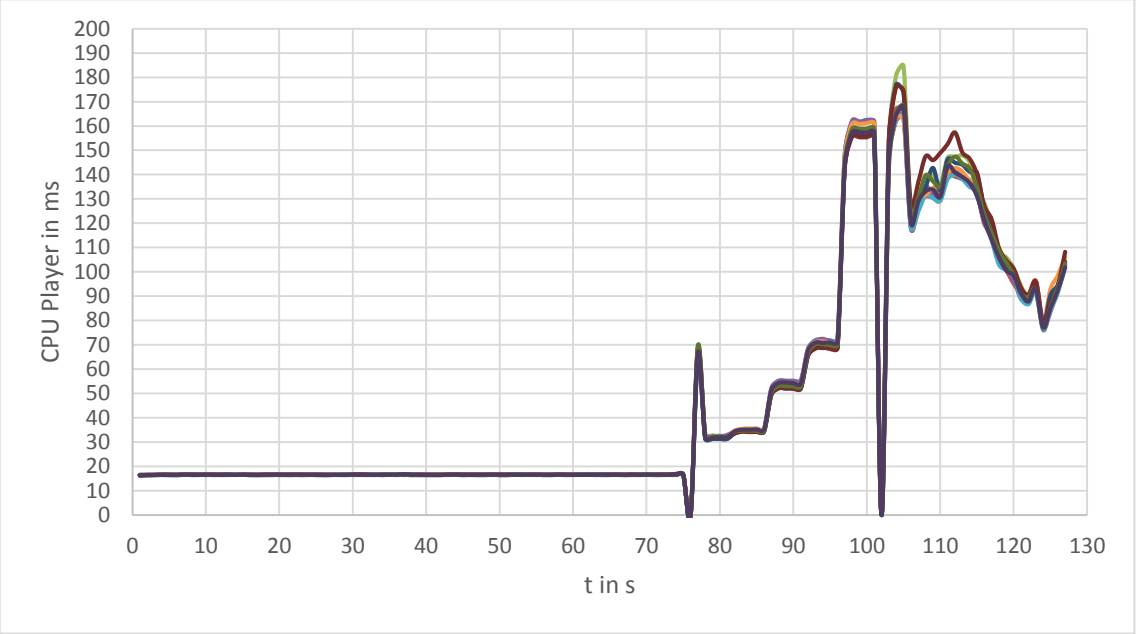

Abbildung 11: CPU-Player-Verlauf

Abbildung 12 visualisiert den Verlauf der *Triangles* aller 10 Tests. Es ist deutlich zu erkennen, dass die Anzahl in jedem Testdurchlauf gleich erfasst wurde.

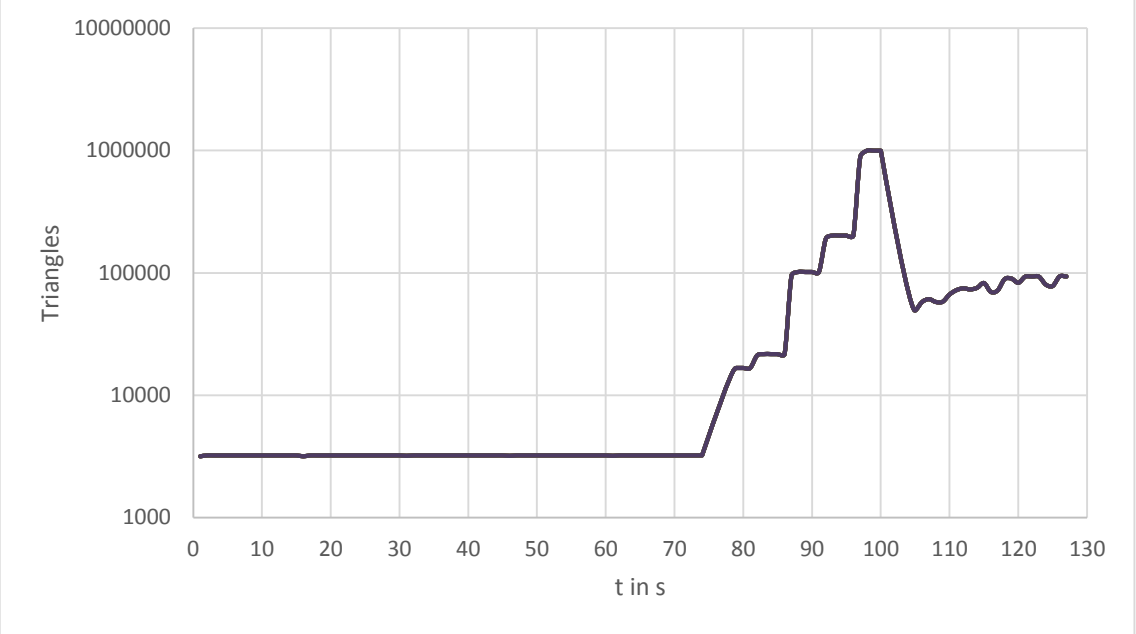

Abbildung 12: Triangle-Verlauf

Abbildung 13 zeigt den Verlauf der *Draw Calls* der Testanwendung. Wie bei den Daten der *Triangles* liegen alle zehn Kurven exakt übereinander.

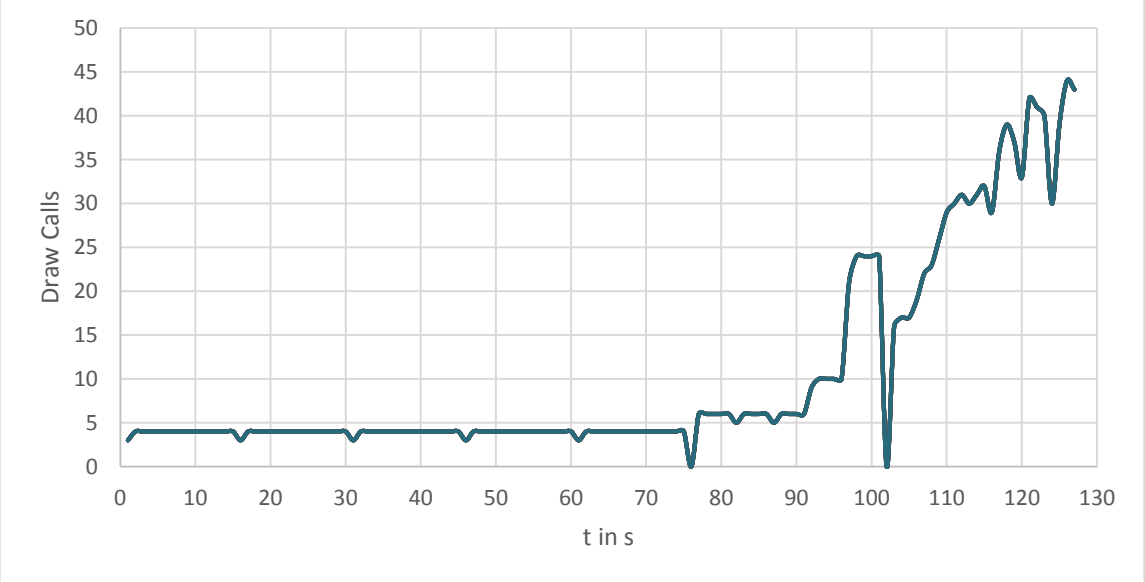

Abbildung 13: Draw Call-Verlauf

Eine statistische Auswertung der Unity-Anwendung ist möglich. Mit steigender Szenenkomplexität können Ungenauigkeiten auftreten. Diese Ungenauigkeiten treten bei Berechnungszeiten auf. Bei der Erstellung der Anwendung vordefinierte Werte wie *Triangles* oder *Draw Calls*, verhalten sich bei allen Testdurchläufen gleich. Bei den *Triangles* gibt es jedoch eine Abweichung gegenüber der ursprünglichen Anzahl und der in den *Profiler-Logfiles* angezeigten. In Kapitel 6.3 wird dieser Sachverhalt genauer beschrieben.

Eine Möglichkeit, den Ungenauigkeiten bei der Zeitmessung entgegenzuwirken, ist, die Auswertung mit Durchschnittswerten aus mehreren Tests durchzuführen. Dies erhöht jedoch die Gesamtdauer des Tests, da jedes zu analysierende Gerät die Anwendung mehrfach durchlaufen muss.

3.4 Beeinflussen die Aufzeichnungssysteme die Daten?

Um zu überprüfen, ob die Aufzeichnungssysteme die Daten beeinflussen, wurde die Testanwendung *3scenes_125seconds_60fps.apk*[83] auf 3 Testsystemen ausgeführt. Diese Systeme sind in Tabelle 1 in Kapitel 2.1 genauer beschrieben.

Für jedes der 3 Testsysteme wurden 10 Datensätze aufgezeichnet. Für die Berechnungen der Standardabweichungen wurden alle 30 Datensätze verwendet. Abbildung 14 zeigt die Verläufe der *Frame Time* aller 30 Datensätze. Wie in Abbildung 10 zu sehen ist, liegen weichen alle Kurven nur minimal voneinander ab. Größere Schwankungen treten bei komplexen Szenen auf. Die Erhöhung der Standardabweichung in komplexen Testabschnitten wurde bereits in Kapitel 0.

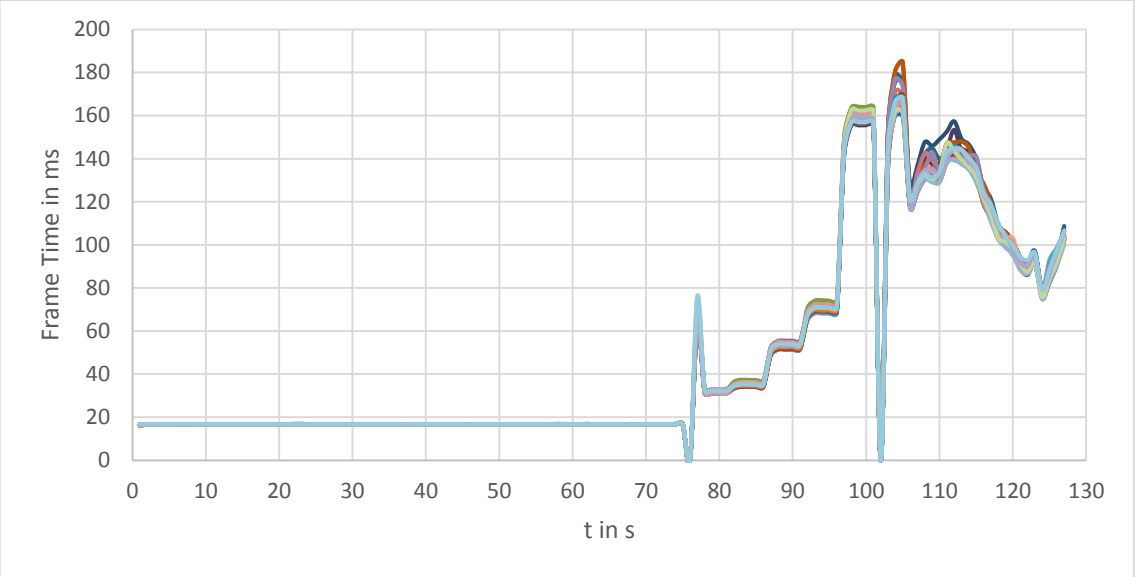

Abbildung 14: Frame Time-Verläufe aller 3 Aufzeichnungssysteme

Im ersten Testabschnitt beträgt die Standardabweichung aller Systeme im Durchschnitt 0,01 ms. Dies entspricht der ermittelten Standardabweichung der Datensätze in Kapitel 3.3. Maximal weichen die Daten 0,19 ms ab und minimal um 0,0 ms.

[83] Anlage CD /apps/3scenes_125seconds_60fps.apk

Im zweiten Testabschnitt weichen die Datenblöcke um Durchschnittlich 1,32ms voneinander ab. Die maximale Abweichung beträgt 3,14 ms und die minimale 0,58 ms. Die durchschnittliche Abweichung ist bei den 3 Systemen um 0,13 ms höher.

Im dritten Testabschnitt liegt die durchschnittliche Abweichung bei 3,166 ms und ist somit niedriger als die in Kapitel 3.3 berechnete Abweichung. Die maximale Abweichung ist mit 6,27 ms niedriger, während die minimale mit 1,54 ms höher ist als bei einem einzelnen System.

Die durchschnittliche Abweichung der Daten über den Kompletten Zeitraum von 125 Sekunden beträgt 0,90 ms. Dieser Wert ist niedriger als die durchschnittliche Abweichung von 0,96 ms, die bei Testsystem 1 berechnet wurde.

Wie bereits bei den für System 1 durchgeführten Tests gibt es Schwankungen lediglich bei den gemessenen Berechnungszeiten. Werte wie *Draw Calls* und *Triangles* sind in jedem Aufzeichnungsdurchgang identisch. Die ermittelten Standardabweichungen der Berechnungszeiten, entsprechen derer in Kapitel 3.3. Daraus folgt, dass beliebige Aufzeichnungssysteme verwendet werden können, ohne die Daten zu beeinflussen.

3.5 Ist eine Verwendung der PBR-Shader auf mobilen Endgeräten möglich?

Die Testanwendung nutzte *PBR-Shader* für alle Objekte und lief auf einem drei Jahre alten *Android*-Gerät. Daraus folgt, dass eine Verwendung der *PBR-Shader* möglich ist. Wie stark die *Shader* die Anwendungsleistung beeinflussen, wird mit der in Kapitel 4 beschriebenen Analyseanwendung mit unterschiedlichen *Android*-Geräten getestet.

3.6 Umwandlung der Unity-Profiler-Daten

In diesem Kapitel wird die Umwandlung der mit *Unity* erstellten *Logfiles* zur statistischen Auswertung beschrieben. Die Inhalte der *Logfiles* werden mit *Python* umgewandelt. Mit dem *Skript* in Anlage 5 werden *Logfiles* in eine *CSV*-Dateien umgewandelt.

Im Folgenden Textabschnitt befindet sich ein Auszug einer mit der Testanwendung erstellten *Logfile*. Jeder *Profiler*-Block enthält die Daten eines Zeitabschnittes von einer Sekunde. Viele Parameter, wie die *Frame Time*, *Draw Calls* sowie die *Triangles* werden mit minimalen, maximalen und durchschnittlichen Werten protokolliert. Zur Auswertung müssen diese gefiltert und in ein für die Tabellenkalkulation lesbares Datenformat umgewandelt werden. Ein Filtern der Daten ist nötig, da jede *Logfile* Elemente wie Trennzeichen, Zeitstempel oder *Android*-Systemmeldungen enthält.

3scenes_125seconds_01.txt[84]

```
07-22 10:26:22.169: D/Unity(14094): Android Unity internal profiler stats:
07-22 10:26:22.169: D/Unity(14094): cpu-player>    min:  0.0   max: 27.9   (…)
07-22 10:26:22.169: D/Unity(14094): cpu-ogles-drv> min:  0.0   max:  0.0   (…)
07-22 10:26:22.169: D/Unity(14094): cpu-present>   min:  0.0   max:  0.9   (…)
07-22 10:26:22.169: D/Unity(14094): frametime>     min:  0.0   max: 28.0   (…)
07-22 10:26:22.169: D/Unity(14094): batches>       min:   0   max:   4   avg:   3
07-22 10:26:22.169: D/Unity(14094): draw calls>    min:   0   max:   4   avg:   3
07-22 10:26:22.169: D/Unity(14094): tris>          min:    0  max:  3218  avg:  3164
07-22 10:26:22.169: D/Unity(14094): verts>         min:    0  max:  6074  avg:  5972
```

3.6.1 Ablauf der Umwandlung

Im Folgenden wird auf die einzelnen Bestandteile des *Python-Skriptes* eingegangen. Die Umwandlung der Daten ist in sechs Schritte unterteilt.

1. **Import von Programmbibliotheken**
2. **Setzen von Ein- und Ausgabepfad**
3. **Definition des Zwischenspeichers**
4. **Lesen der Logfile**
5. **Schreiben der Logfile in den Zwischenspeicher**
6. **Ausgabe der Datenbank in eine CSV-Datei**

[84] Anlage CD /3scenes_125seconds/3scenes_125seconds_01.txt

3.6.2 Import von Programmbibliotheken

Zu Beginn des *filereader_v17.py-Skript*s in Anlage 5 werden einige *Python*-Module geladen. Diese stellen zusätzliche Funktionen zur Verfügung. Module sind Erweiterungen die den Funktionsumfang einer *Python*-Anwendung erhöhen.[85]

```
import re, sys, os, collections
```

Das *re*-Modul importiert die *Regular Expression Operations*. Diese ermöglichen es eine Zeichenfolge anhand vorher definierter Parameter zu analysieren.[86] Im Verlauf der Umwandlung der *Logfiles* werden damit Zeichenfolgen voneinander getrennt. Mit dem *sys*-Modul kann auf zusätzliche Eingabevariablen zugegriffen werden. Hierdurch ist es möglich, beliebige externe Variablen anzusprechen. Im vorliegenden *Skript* sind dies die *Unity-Logfiles* und das *Python-Skript* selbst. Durch den Import des *os*-Moduls kann auf das Dateisystem des Betriebssystems zugegriffen werden.[87] Dies wird für die Ein- und Ausgabe von Dateien benötigt. Das *collections*-Modul ermöglicht das Generieren von Containerobjekten.[88] Diese Container können mit anderen Inhalten, beispielsweise Listen von Variablen, gefüllt werden.

3.6.3 Setzen von Ein- und Ausgabepfad

Im folgenden Abschnitt des *Skript*s werden die Ein- und Ausgabepfade der Daten definiert.

```
input_file = sys.argv[1]
input_file = os.path.basename(input_file)
print "opened logfile as " + input_file
output_file = os.path.basename(input_file.replace(".txt","_converted.csv"))
```

[85] Vgl. HETLAND, 2008: 17

[86] Vgl. PYTHON DOCUMENTATION, https://docs.python.org/2/library/re.html?highlight=re#module-re, 16.10.15

[87] Vgl. PYTHON DOCUMENTATION, https://docs.python.org/2/library/os.html?highlight=os#module-os, 16.10.15

[88] Vgl. PYTHON DOCUMENTATION, https://docs.python.org/2/library/collections.html?highlight=collections#module-collections, 16.10.15

Die zu bearbeitende *Logfile* wird vom *Skript* gelesen. Durch das vorher geladene *sys*-Modul ist die Eingabe mehrerer Argumente möglich. Über die erste Zeile, des dargestellten Abschnittes, werden die *Unity-Logfile* sowie das *Python-Skript* als Argumente gelesen.[89] Mithilfe des *Skripts* in Anlage 4 wird dieser Prozess erweitert. Die *Batch*-Datei *convert_logfiles.bat* ermöglicht es mittels *Drag and Drop* mehrere *Logfiles* auf einmal zu konvertieren.

Der Dateiname wird mit *Python* ausgelesen. Hierfür wird das zuvor importierte *os*-Modul verwendet. Zu Analysezwecken wird der Dateiname der aktuellen *Logfile* ausgegeben. Im Anschluss wird die Ausgabedatei definiert. Deren Name leitet sich von der Eingabedatei ab. Der Dateityp wird von einer Text- zu einer *CSV*-Datei geändert. Im selben Schritt wird der Dateiname um den Suffix *_converted* erweitert. Hierdurch wird deutlich, dass es sich um die konvertierte *Logfile* handelt.

3.6.4 Definition des Zwischenspeichers

Die eingelesenen *Logfiles* werden in einer Datenbank zwischengespeichert. Diese Datenbank wird innerhalb einer Funktion mit dem Namen *create_dict* generiert. Die Erstellung der Datenbank wurde in eine Funktion ausgelagert, da dies die Übersichtlichkeit bei der Bearbeitung des *Python-Skripts* erhöht.

```
def create_dict(my_dict):
    print "creating dict"
    my_dict['frametime']=collections.OrderedDict()
    my_dict['frametime']['min']=[]
    my_dict['frametime']['max']=[]
    my_dict['frametime']['avg']=[]
    my_dict['draw calls']=collections.OrderedDict()
    my_dict['draw calls']['min']=[]
    my_dict['draw calls']['max']=[]
    my_dict['draw calls']['avg']=[]
   (...)
```

Hierbei wird auf die *Ordered Dictionary*-Funktion des *collections*-Moduls zurückgegriffen. Diese ermöglicht die Erstellung von Datenbanken mit vordefinierter Struktur. Ein *Dictionary* besteht aus einem Schlüssel und dazugehörigen Werten.[90] Da zu jedem Datenbankeintrag mehrere Unterpunkte gehören werden diese ebenfalls mittels *Ordered Dictionary* innerhalb eines Datenbankeintrages erstellt. Ohne Verwendung des *Ordered*

[89] Vgl. HETLAND, 2008: 222
[90] Vgl. HETLAND, 2008: 70

Dictonary gäbe es keine feste Struktur innerhalb der Datenbank. Die Werte wären zwar unter den korrekten Haupt- und Nebenschlüsseln in der korrekten Reihenfolge gespeichert. Die Reihenfolge der Schlüssel und ihrer jeweiligen Nebenschlüssel kann jedoch variieren. Ohne ein *Ordered Dictionary* kann die Datenbank folgendermaßen aussehen:

```
Schlüssel 3
        Nebenschlüssel 2
                Wert 1
                Wert 2
        Nebenschlüssel 1
                Wert 1
                Wert 2
Schlüssel 9
        Nebenschlüssel 3
                Wert 1
                Wert 2
```

Durch die Verwendung des *Ordered-Dictonaries* ergibt sich folgende Datenbankstruktur:

```
Schlüssel 1
        Nebenschlüssel 1
                Wert 1
                Wert 2
        Nebenschlüssel 2
                Wert 1
                Wert 2
Schlüssel 2
        Nebenschlüssel 1
                Wert 1
                Wert 2
```

3.6.5 Lesen der Logfile

Nachdem die Datenbank generiert wurde, liest das *Skript* jede Zeile der *Logfile* aus.

```
with open(input_file) as logfile:
    scenecount = 1
    for line in logfile:
        current_line = line.rstrip()
        if "changing level to:" in current_line:
            scenecount = scenecount +1
            for k, v in my_dict.items():
                for subk, subv in v.items():
                    subv.append("----------")
        else:
            if ">" in current_line:
                process_line(current_line, my_dict)
```

Mit einer Wenn-Abfrage wird überprüft, ob die aktuelle Zeile den Ausdruck *changing level to*: enthält. Dieser Ausdruck wird vom verwendeten *change_scene.cs-Skript* generiert. Wenn dies der Fall ist, wird eine Reihe von Bindestrichen in alle Werteblöcke der Datenbank geschrieben. Diese dienen bei der statistischen Auswertung als Markierung für den Szenenwechsel und werden ebenfalls in die *CSV*-Datei geschrieben. Alle für die Auswertung relevanten Zeilen enthalten innerhalb der *Logfile* ein Größer-als-Zeichen. Mit einer Wenn-Funktion werden die relevanten Zeilen herausgefiltert. Diese werden anschließend an eine Funktion mit der Bezeichnung *process_line* weitergegeben.

3.6.6 Schreiben der Logfile in den Zwischenspeicher

Die Funktion *process_line* beinhaltet das Auslesen der verschiedenen Schlüssel, Nebenschlüssel und Werte sowie das Schreiben dieser in die vordefinierte Datenbank.

```
def process_line(current_line, my_dict):
    key_value_list=current_line.lstrip().split(">")
    key_name = key_value_list[0]
    key_name_list = key_name_pattern.findall(key_name)
```

Die aktuelle Zeile der *Logfile* wird anhand des Größer-als-Zeichens in zwei Listenpunkte aufgetrennt. Dies geschieht über die Funktionen *lstrip()* und *split()*.Um sicher zu stellen, dass keine Leerzeichen am Anfang der Zeile vorhanden sind werden diese mittels *lstrip*() entfernt.[91] Über *split(„>“)* wird definiert, dass die Zeichenfolge am Größer-als-Zeichen geteilt wird.[92] Der Textblock links vom Größer-als-Zeichen wird später genutzt, um eine Verbindung zum Hauptschlüssel der Datenbank herzustellen. Der rechte Abschnitt beinhaltet die Nebenschlüssel mit den dazugehörenden Werten. Über die *compile*-Funktion des *re*-Moduls wird anhand des vorher definierten *key_name_pattern* der linke Abschnitt der Zeichenfolge anhand des Doppelpunktes vor dem Hauptschlüsselnamen aufgetrennt. Dadurch entfallen Zeitstempel und Dateipfad der Anwendung auf dem *Android*-Gerät. Im folgenden Abschnitt ist eine *Logfile*-Zeile vor und nach der Bearbeitung zu sehen:

```
05-13 14:40:56.230: D/Unity(9783): frametime>     min:  0.0   max: 253.5   avg: 66.9
```

[91] Vgl. HETLAND 2008, 64

[92] Vgl. HETLAND 2008, 63

Nach der Bearbeitung sind die beiden Blöcke voneinander getrennt und Bestandteile zweier Listen. Der Hauptschlüssel befindet sich in der Liste mit der Bezeichnung *key_name_list* als Wert 1 und die Nebenschlüssel mit den Werten in der *key_value_list* als Wert 1.

```
frametime
min:  0.0   max: 253.5   avg: 66.9
```

Im nachfolgenden Abschnitt wird der Hauptschlüsselname aus der *key_name_list*-Liste ausgelesen, hierfür wird auf den Wert 1 der Liste zugegriffen. Dieser wird als *key_name* zwischengespeichert. Die Nebenschlüssel und Werte, welche als Wert 1 in der Liste *key_value_list* vorliegen, werden zur Weiterverarbeitung als *key_values* abgelegt.

```
for elements in key_name_list:
    key_name = elements[1]
key_values = key_value_list[1]
```

An diesem Punkt sind alle Nebenschlüssel und Werte in einer Liste mit dem Namen *key_values* vorhanden. Der folgende Abschnitt beschreibt das Auftrennen dieser Liste in einzelne Blöcke. Jeder Block definiert einen Nebenschlüssel und den dazugehörenden Wert.

Mithilfe der *compile*-Funktion *re*-Moduls werden die vorher abgelegten *key_values* in einzelne Datenblöcke unterteilt. Hierbei wird jeder Nebenschlüssel mit dem dazugehörigen Wert als Listenpunkt abgelegt.

```
match = pattern.findall(key_values)
```

Grundlage hierfür ist ein vorher definiertes Muster. Dieses wird zu Beginn des *Skript*s als *pattern* definiert.

```
pattern = re.compile("\s+([a-zA-Z\s-]+):?\s+((-*[0-9]+\s..\s-*[0-9]+)|(-*[0-9]+\.*-*[0-9]*))")
```

Mit diesem Muster wird der *key_value* analysiert und in einzelne Listenelemente unterteilt. Der folgende Textabschnitt zeigt den aktuellen *key_value*-Eintrag vor der Bearbeitung:

```
min:  0.0   max: 253.5   avg: 66.9
```

Nach der Bearbeitung sind alle drei Nebenschlüssel und ihre dazugehörenden Werte einzelne Einträge in einer Liste. Diese Liste trägt die Bezeichnung *match*. Innerhalb jedes Listeneintrages befindet sich ein Wertepaar aus Nebenschlüssel und dem dazugehörenden Wert.

```
min:  0.0
max: 253.5
avg: 66.9
```

Im Anschluss durchläuft eine Schleife die Liste *match*. Mit dieser Schleife wird jeder Nebenschlüssel der Datenbank durchlaufen. Es wird überprüft, ob der aktuelle Nebenschlüssel der Datenbank und ein Nebenschlüssel-Eintrag der Liste *match* übereinstimmen. Falls dies zutrifft, wird der Wert in das *Dictionary* geschrieben. Der nachfolgende Auszug des *Python-Skripts* speichert die einzelnen *Logfile*-Informationen in der Datenbank ab:

```
if key_name not in my_dict:
    return False
for each_group in match:
    if each_group not in match:
        return False
    sub_key_name = each_group[0]
    sub_key_name = sub_key_name.rstrip()
    #write subkey values to the subdictionary
    sub_key_value = each_group[1]
    if "frametime" in current_line:
        if sub_key_value == str(0.0):
            sub_key_value = sub_key_value
        else:
            fps = 1000 / float(sub_key_value)
            fps = round(fps, 2)
            sub_key_value = str(fps)
    if ".." in sub_key_value:
        #checks if 2 values exist, happens in fixed update count
        #saves only the second value to the dictionary
        sub_key_value = sub_key_value.split(" .. ")[1]
        my_dict[key_name][sub_key_name].append(sub_key_value)
    else:
        my_dict[key_name][sub_key_name].append(sub_key_value)
```

Um Abstürze des *Skript*es zu verhindern gibt es einen Abfrageblock der, Falls er zutrifft, die *process_line*-Funktion verlässt und mit der nächsten Zeile fortfährt. Dies wird über die *return*-Funktion durchgeführt. Dies verhindert Abstürze im Fall, dass ein Haupt- oder Nebenschlüssel sich nicht in der Datenbank befindet.

3.6.7 Ausgabe der Datenbank in eine CSV-Datei

Für die Ausgabe der Datenbank in eine *CSV*-Datei wird auf eine selbstdefinierte Funktion zurückgegriffen. Diese trägt den Namen *dict_to_csv* und wird ausgeführt, nachdem die *Logfile* in die Datenbank eingetragen wurde. Beim Start der Funktion werden zwei Variablen an die Funktion weitergereicht. Diese sind die am Anfang definierte Ausgabedatei und die Datenbank, in die alle Werte der *Logfile* eingelesen wurden.

```
dict_to_csv(output_file, my_dict)
```

Innerhalb der Funktion wird die Ausgabedatei zunächst geöffnet. Hierdurch ist es möglich, Inhalte in diese zu schreiben.

```
def dict_to_csv(output_file, my_dict):
    with open(output_file, "w") as converted_csv:
```

Das Schreiben der Daten in die Datei erfolgt zeilenweise. Der Schreibvorgang läuft in vier Schritten ab. Im ersten werden die Hauptschlüssel in eine Liste eingetragen. Dies muss unter Berücksichtigung der Nebenschlüssel erfolgen. Denn diese definieren, wie viele freie Spalten zwischen jedem Hauptschlüssel liegen müssen. Im Anschluss werden die Nebenschlüssel ebenfalls in eine Liste geschrieben. Im dritten Schritt werden die Datensätze ebenfalls in einzelne Listen eingetragen. Abschließend werden alle Listen in die *CSV*-Datei geschrieben. Der folgende Quelltext erstellt die Liste der Hauptschlüssel:

```
csv_headers = my_dict.keys()
key_number = 0
for key in csv_headers:
    key_number = key_number +1
    if key in my_dict:
        #sets subkey counter to zero for every key block
        count = 0
        for value in my_dict[key]:
            count = count +1
        insert = 1
        while insert < count:
            csv_headers.insert(key_number, "---")
            insert = insert + 1
    else:
        False
```

Da zu jedem Haupt- mehrere Nebenschlüssel gehören muss die Anzahl der freien Spalten zwischen den Hauptschlüsseln ermittelt werden. Um dies zu ermitteln, wird jeder Hauptschlüssel auf die Anzahl der Nebenschlüssel überprüft. Hierfür wird eine Schleife mit integriertem Zähler genutzt. Alle freien Spalten rechts vom aktuellen Hauptschlüssel

werden anschließend mit einer Zeichenfolge von drei Bindestrichen gefüllt. Dies geschieht solange, bis die Anzahl der Nebenschlüssel erreicht ist. Es ist zu beachten, dass unter den ersten Hauptschlüssel bereits ein Nebenschlüssel geschrieben wird. Deshalb muss die Anzahl der Blöcke, die mit Bindestrichen gefüllt werden, um 1 verringert werden.

Die Nebenschlüssel werden in eine Liste mit der Bezeichnung *csv_subheaders* geschrieben. Hierfür werden alle Nebenschlüssel der Datenbank gelesen und einer Liste hinzugefügt.

```
csv_subheaders=[]
for k, v in my_dict.iteritems():
   for subk, subv in v.iteritems():
      #gets the subheader name from the subk of the nested dictionary
      #appends the subk at the end of the subheaders list
      csv_subheaders.append(subk)
```

Im folgenden Abschnitt werden die Haupt- und Nebenschlüssel in die *CSV*-Datei geschrieben. Zuerst werden die Hauptschlüssel geschrieben, indem die Liste *csv_headers* mit einer Schleife durchlaufen wird. Jeder Listeneintrag wird hintereinander in die *CSV*-Datei geschrieben. Getrennt werden die einzelnen Einträge von Tabulatoren. Diese Tabulatoren dienen später in der statistischen Auswertung als Trennzeichen zwischen den Spalten. Vor dem ersten Hauptschlüssel befindet sich ebenfalls ein Tabulator. Dieser sorgt dafür, dass die Hauptschlüssel erst in Spalte 2 beginnen. Grund hierfür ist der im zweiten Schritt, vor den Nebenschlüsseln, eingefügte Datenblock *Data*, welcher als Überschrift für die Anzahl der Datenblöcke fungiert. Jeweils nach dem Schreiben der Haupt- und Nebenschlüssel wird am Ende jeder geschriebenen Zeile ein Zeilenumbruch erzwungen.

```
#writes headers and subheaders
converted_csv.write("\t")
for header in csv_headers:
   converted_csv.write(header + "\t")
converted_csv.write("\n" + "Data" +"\t")
for subheader in csv_subheaders:
   converted_csv.write(subheader + "\t")
converted_csv.write("\n")
```

Um die zu den Nebenschlüsseln gehörenden Werte schreiben zu können, müssen diese in eine Form gebracht werden, welche ein zeilenweises Ausgeben ermöglicht. Innerhalb der Datenbank sind alle Werte als Liste zum passenden Nebenschlüssel abgelegt. Um

die Werte zeilenweise schreiben zu können, müssen alle Werte eines jeweiligen Datenblocks in eine Liste gebracht werden.

```
list_subv = []
for k, v in my_dict.iteritems():
    for subk, subv in v.iteritems():
        list_subv.append(subv)
```

Im ersten Schritt dieser Umwandlung wird eine leere Liste erstellt. Diese Liste mit dem Namen *list_subv* dient als Zwischenspeicher für alle Wertelisten. Über eine Schleife wird anschließend jede Werteliste zu dieser hinzugefügt. Dadurch entsteht eine mit Listen gefüllte Liste. Der Aufbau der Liste *list_subv* ist auszugweise in Tabelle 3 dargestellt. Die einzelnen Listen innerhalb von *list_subv* sind abwechselnd grau und weiß hinterlegt.

Über die *map* Funktion werden die Listen innerhalb von *list_subv* so transponiert, dass alle Werte eines Datenblocks in jeweils einer Liste sind. Die Werte werden mittels einer Transposition um die Hauptachse der Tabelle getauscht.

```
list_subv = map(None,*list_subv)
```

Der Ablauf der Transposition ist in den folgenden Tabellen dargestellt. Die in Tabelle 3 dargestellten Listen zeigen die aktuelle Datenstruktur der Haupt und Nebenschlüssel.

Frame Time	**---**	**---**	**Draw Calls**	**---**	**---**
min	**max**	**avg**	**min**	**max**	**avg**
0	27,5	16,7	0	4	3
16,2	17,1	16,7	4	4	4
13,3	20,1	16,7	4	4	4
16,3	18,6	16,7	4	4	4
16,4	17	16,7	4	4	4
13,6	19,8	16,7	4	4	4

Tabelle 3: Tabelle vor der Transponierung

Tabelle 4 zeigt die Hauptachse an welcher die Transponierung der Daten durchgeführt wird in grau.

Frame Time	---	---	**Draw Calls**	---	---
min	**max**	**avg**	**min**	**max**	**avg**
0	27,5	16,7	0	4	3
16,2	17,1	16,7	4	4	4
13,3	20,1	16,7	4	4	4
16,3	18,6	16,7	4	4	4
16,4	17	16,7	4	4	4
13,6	19,8	16,7	4	4	4

Tabelle 4: Hauptachse der Transponierung

In Tabelle 5 wurde die Transposition durchgeführt. Die Listen, welche abwechselnd grau und weiß hinterlegt sind, enthalten nun nicht mehr alle Werte eines Nebenschlüssels. Stattdessen beinhaltet jede Liste nun alle Werte eines *Profiler*-Blocks. Diese transponierten Daten können zeilenweise ausgegeben werden.

Frame Time	---	---	**Draw Calls**	---	---
min	**max**	**avg**	**min**	**max**	**avg**
0	27,5	16,7	0	4	3
16,2	17,1	16,7	4	4	4
13,3	20,1	16,7	4	4	4
16,3	18,6	16,7	4	4	4
16,4	17	16,7	4	4	4
13,6	19,8	16,7	4	4	4

Tabelle 5: Transponierte Daten

Im folgenden Abschnitt werden die Werteblöcke in die *CSV*-Datei geschrieben. Beim Durchlauf jeder Zeile wird der Zähler für die Datenblockzahl erhöht. Um den Szenenwechsel für die statistische Auswertung eindeutig zu markieren, wird dieser mit jeweils

drei Bindestrichen pro Spalte ausgegeben. Zur besseren Lesbarkeit wird bei der Ausgabe das Trennzeichen zwischen Vor- und Nachkommstelle getauscht. Der Punkt, welcher in den *Logfile*s als Trennzeichen fungiert wird, gegen ein Komma getauscht.

```
    for subv in list_subv:
        #writes the current datablock number and increases the counter
        #tabstop as a seperator between the values
        if "----------" in subv:
            converted_csv.write("---" + "\t")
            counter = counter + 0
        else:
            converted_csv.write(str(counter) + "\t")
            counter = counter +1
        for value in subv:
            if value is None:
                converted_csv.write(str(value) + "\t")
            else:
                converted_csv.write(str(value.replace(".", ",")) + "\t")
        converted_csv.write("\n")
    #prints the processed datablocks and closes the csv
    print "\nprocessed", scenecount, "scenes"
    print "processed", counter -1, "values \n"
    converted_csv.close()
print "saved csv as " + output_file
```

Die Ausgabe der Blöcke erfolgt nach demselben Prinzip wie bei den Haupt- und Nebenschlüsseln. Um zu überprüfen, ob alle zu erwartenden Datenblöcke geschrieben wurden, werden zusätzlich noch die Blöcke gezählt. Die Anzahl dieser wird nach Abschluss des *Skript*es angezeigt. Der Erfolgreiche Durchlauf des *Skript*es wird mit der Meldung *saved csv as* und dem dazugehörenden Dateinamen angezeigt.

3.7 Fazit

Alle Fragen die in Kapitel 2.2 definiert wurden konnten beantwortet werden. Anhand des Testszenarios wurde belegt, dass es möglich ist statistische Daten für die Leistungsanalyse mithilfe einer *Unity*-App zu ermitteln. Zunächst gab es Probleme bei der Aufzeichnung der *Profiler*-Daten. Diese wurden fehlerhaft übertragen, sobald die *Frame Time* der Anwendung von der *Frame Time* des *Android*-Systems, welche 16,7 ms beträgt, abwich. Um dieses Problem zu lösen, wurde das *Rendern* von 60 fps in der 3D-Anwendung erzwungen, denn dies ist die Bildrate mit der ein aktuelles *Android*-System läuft. Die über den *Unity-Profiler* aufgezeichneten Daten sind reproduzierbar und nicht vom Aufzeichnungssystem abhängig. Die aufgezeichneten Daten können unbearbeitet jedoch nicht statistisch analysiert werden. Hierfür müssen sie mithilfe eines *Skriptes* umgewandelt werden.

4 Analyseanwendung

Durch die Testdurchläufe wurde geklärt, dass eine Leistungsanalyse möglich ist. Im folgenden Kapitel wird eine Anwendung für die statistische Auswertung beschrieben.

Zunächst werden Forschungsfragen definiert, welche mit der Analyseanwendung geprüft werden sollen. Mithilfe dieser Forschungsfragen wird eine Testanwendung konzipiert. Anhand der aus dieser Anwendung ermittelten empirischen Daten wird exemplarisch eine statistische Analyse für diverse *Android*-Geräte durchgeführt.

4.1 Definition der Forschungsfragen

1. Gibt es Leistungsunterschiede zwischen einem Objekt mit hohem Detailgrad und vielen Objekten mit niedrigerem Detailgrad?

2. Steigt der Leistungsbedarf bei der Verwendung vieler detaillierter 3D-Modelle linear an?

3. Wie viele Lichtquellen können auf einem spezifischen Android-Gerät parallel aktiv sein, um eine flüssige Bildrate zu ermöglichen?

4. Wie verhalten sich Android-Geräte verschiedener Generationen mit vielen sichtbaren animierten Charakteren in einer Szene?

5. Ist der Leistungsunterschied zwischen Legacy- und PBR-Shadern auf einem modernen Android Gerät geringer?

7. Verhalten sich Geräte verschiedener Hersteller, die über dieselbe Hardware verfügen, in bestimmten Szenarien gleich?

4.2 Konzeption der Anwendung

Anhand der im letzten Abschnitt beschriebenen Forschungsfragen wird eine Anwendung konzipiert. Diese Anwendung gliedert sich in fünf Testblöcke. Im folgenden Abschnitt wird darauf eingegangen, wie die einzelnen Forschungsfragen geprüft werden.

4.2.1 Testblock 1

Um die Auswirkung von eines steigenden Detailgrades auf die Geräteleistung zu prüfen wird, die Anzahl der gerenderten *Triangles* in einer Testszene stetig erhöht. Hierfür wird ein 3D-Modell in mehreren Detailstufen verwendet. Diese steigen von 10.000 *Triangles* bis 1.000.000 *Triangles* an. Bis 100.000 *Triangles* steigt die Zahl jener in 10.000er Schritten. Die nachfolgenden Tests werden in bis 500.000 *Triangles* in 25.000er Schritten durchgeführt. Abschließend werden Charaktermodelle mit 750.000 und 1.000.000 *Triangles* geladen.

Der Test ist eine erweiterte Variante aus Szene 2 der Testanwendung. Die Abstufungen der Detailgrade sind feiner.

4.2.2 Testblock 2

In diesem Abschnitt wird vor allem auf die *Draw Calls* in Verbindung mit der Objektanzahl eingegangen.

Zum Testen der Objektanzahl werden nacheinander gleiche Objekte geladen. Es ist wichtig, dass in diesem Test nicht auf *Batching* zurückgegriffen wird. Da *Static-Batching* manuell für jedes Objekt aktiviert werden muss, funktioniert es standardmäßig nicht. Um *Dynamic-Batching* zu verhindern müssen Objekte entweder verschiedene Skalierungen oder Materialien besitzen. Objekte, die dynamische Schatten empfangen, werden durch das *Batching* ebenfalls nicht zusammengefasst.[93] Diese dynamischen Schatten wirken sich jedoch negativ auf die Anwendungsleistung aus. Für die bestmögliche Grafikqualität sind dynamische Schatten jedoch wichtig, weshalb nicht auf sie verzichtet wird. Da für *AR*-Anwendungen 3D-Objekte in reale Umgebungen projiziert werden müssen Schattenverläufe dynamisch berechnet werden.

[93] Vgl. UNITY, http://docs.unity3d.com/Manual/DrawCallBatching.html, 14.11.15

Die Anwendung durchläuft den ersten Abschnitt von Testblock 2 insgesamt fünfmal. Bei jedem Durchlauf wird die Anzahl der *Triangles* pro Objekt erhöht. Hierbei werden Tests von 10.000 bis 50.000 *Triangles* durchgeführt.

Die Anzahl der Objekte wird von 1 auf bis zu 50 erhöht. Die 3D-Modelle 1 bis 20 werden einzeln eingeblendet. Da zu erwarten ist, dass das *Rendern* von vielen Charakteren auf älteren *Android* Geräten sehr lange dauert, werden diese Objekte blockweise zu je 5 Objekten, eingeblendet. Das einzelne Einblenden jedes Charakters würde aufgrund der erzwungenen 60 fps auf langsameren Geräten eine sehr lange Aufzeichnungsdauer der Daten zur Folge haben.

Die Tests basieren ebenfalls auf Szene 2 der Testanwendung. Mit dem Unterschied, dass nacheinander mehr Objekte geladen werden. Hierfür musste das *Skript*, welches die Objekte ein- und ausblendet, umgeschrieben werden. Das *change_objects.cs-Skript* blendete Objekte gleichzeitig ein und aus. Das neue *load_objects.cs-Skript* blendet die vorhergehenden Objekte nicht aus. Dadurch werden immer mehr Objekte in eine Szene geladen, bis die gewünschte Anzahl erreicht ist.

4.2.3 Testblock 3

In diesem Abschnitt der Anwendung wird die Leistungsauswirkung von Materialien erfasst. Der *Standard Shader* in Unity 5 ist aufgrund der physikalisch korrekten Berechnungen leistungshungriger als einer der für mobile Geräte optimierten *Mobile Shader*.[94] Um den Unterschied zwischen dem für mobile Geräte optimierten und dem *PBR-Shader* zu bestimmen, wird Testblock 1 mit den *PBR-Shadern* erneut durchlaufen. Mit den Resultaten aus Testblock 3 wird geprüft, welche Art von *Shadern* sich für bestimmte *Android*-Endgeräte eignet.

4.2.4 Testblock 4

Dieser Abschnitt der Testanwendung prüft die Leistungsfähigkeit bei der Berechnung von Lichtquellen. Die Szene wird dreimal durchlaufen. Es werden nacheinander immer mehr Lichtquellen geladen. Im ersten Durchlauf wird keine der Lichtquellen Schatten

[94] Vgl. UNITY, http://docs.unity3d.com/Manual/shader-StandardShader.html, 14.11.15

werfen. Der zweite Durchlauf wird mit harten Schatten durchgeführt. Im dritten Teilabschnitt werden die Lichtquellen weiche Schatten werfen.

4.2.5 Testblock 5

Der letzte Testabschnitt ähnelt Testblock 2. Es werden nacheinander bis zu 50 animierte Charaktermodelle geladen. Mit diesem Test wird die Leistungsfähigkeit bei der Berechnung von deformierten Modellen erfasst.

4.2.6 Test von Renderpfaden

Die Geschwindigkeit des *Renderings* in *Unity* hängt vom verwendeten *Renderpfad* ab. Unity verfügt über zwei *Renderpfade*. Diese sind *Deferred Rendering* und *Forward Rendering*. Beide skalieren unterschiedlich der Anzahl von 3D-Modellen und Lichtquellen.

Forward *Rendering* ist ein weit verbreitetes Verfahren. Einzelne 3D-Objekte werden von der *GPU* nacheinander geladen, mit Lichtinformationen versehen und abschließend als Bild ausgegeben.[95] Dies erfolgt getrennt für jede Geometrie Bei *Forward Rendering* ist die Anzahl der in voller Qualität gerenderten Lichtquellen in *Unity* vom *Pixel Light Count* abhängig. Liegt die Anzahl der Lichtquellen über dem *Pixel Light Count,* werden Lichtquellen nicht mehr pro Pixel berechnet. Stattdessen werden nur noch die *Vertices* eines Objektes mit Lichtinformationen versehen.

Das *Deferred Rendering* eignet sich am besten für viele in Echtzeit berechnete Lichtquellen, benötigt jedoch Hardware, die es dies unterstützt.[96] Wie beim *Forward Rendering* werden alle Geometrien nacheinander abgearbeitet. Jedoch auf die Lichtberechnung der einzelnen Geometrien verzichtet. Die Berechnung der Beleuchtung erfolgt verzögert (*deferred*) für alle 3D-Objekte gemeinsam.[97] Jede Lichtquelle wird beim *Deferred Rendering* pro Pixel berechnet, dies sorgt für eine höhere Genauigkeit der Lichtberechnungen und resultiert in einem Bild.

---

[95] Vgl. OWENS, http://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342, 01.11.15

[96] Vgl. UNITY, http://docs.unity3d.com/Manual/RenderingPaths.html, 01.12.15

[97] Vgl. OWENS, http://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342, 01.11.15

Um zu überprüfen wie gut ein *Android* Endgerät mit verschiedenen *Renderpfaden* skaliert wird muss die Gesamte Testanwendung zweimal durchlaufen werden. Einmal mit *Deferred* und im zweiten Durchlauf mit *Forward Rendering.*

Zusätzlich ist zu beachten, dass *Deferred Rendering* bei *Android* Geräten nur mit Unterstützung von *OpenGL ES 3.0 & MRT* möglich ist[98]. Geräte die lediglich *Open GL ES 2.0* unterstützen, wie beispielsweise das für den ersten Test genutzte *Samsung Galaxy S3* wählen automatisch den niedrigeren Grafikmodus.

[98] Vgl. UNITY, http://docs.unity3d.com/Manual/RenderingPaths.html, Rendering Paths Comparison, 14.09.15

4.3 Aufbau der Analyse-Anwendung

4.3.1 Testblock 1

Diese Unity-Szene besteht aus insgesamt 28 Charaktermodellen. Die Szene trägt die Bezeichnung testblock_01_01. Alle 3D-Modelle basieren auf demselben Grundmodell. Als Testobjekt wird ein Charakter aus den *Autodesk 3ds Max 2015 Sample Files*[99] verwendet. In dieser Szene wird der Charakter *CMan0002-M3-CS.max* verwendet. Alle 28 Charaktermodelle sind in der Objektliste des *switch_objects.cs-Skriptes* hinterlegt.

Es werden die optimierten *Legacy Mobile Shader* von Unity für die Charaktere sowie die Umgebung verwendet. Jeder Charakter besitzt drei Materialien mit jeweils zwei *Texturen*. Alle drei sind keine *PBR*-Shader. Für den Körper und den Kopf wird jeweils der *Mobile/Bumped Diffuse Shader* genutzt. Jedes Material besitzt eine *Diffuse*- und eine *Normalmap*. Für die Haare des Charakters wird der *Legacy/Transparent Bumped Diffuse Shader* genutzt.

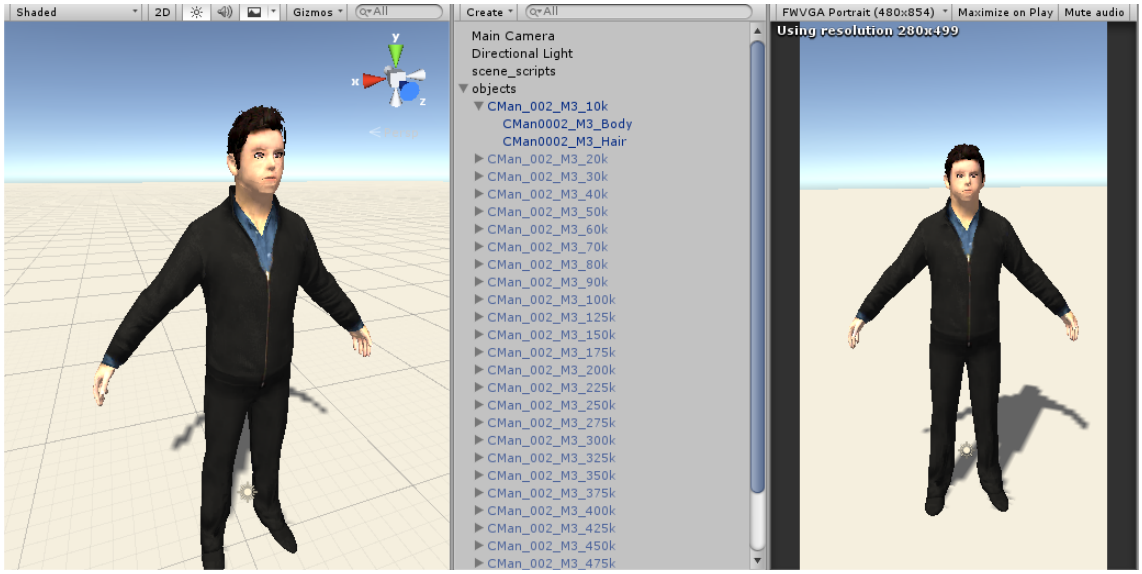

Abbildung 15: Aufbau von Testblock 1

Als Beleuchtung kommen ein direktionales Licht mit hartem Schattenwurf, welche die Sonne darstellt, sowie ein einfarbiger *Skydome* zum Einsatz.

---

[99] Vgl. 3DS MAX 2015 SAMPLE FILES, http://knowledge.autodesk.com/support/3ds-max/downloads/caas/downloads/content/3ds-max-2015-sample-files.html, 01.11.15

4.3.2 Testblock 2

Jeder der 5 Abschnitte besitzt jeweils 50 identische Charaktermodelle. Abbildung 16 zeigt die 50 Modelle des ersten Abschnitts, jedes der gezeigten Modelle besteht aus 10.000 *Triangles*. Der Charakter ist derselbe wie auch in Testblock 1. Die Charaktere werden über das *load_objects.cs-Skript* gesteuert. Alle Charaktere eines Szenenabschnitts befinden sich jeweils in einer Objektliste. Die 5 Abschnitte sind jeweils in eigenen *Unity*-Szenen. Die Testabschnitte sind von testblock_02_01 bis testblock_02_05 durchnummeriert. Die Materialien sowie die Beleuchtung der Szene entsprechen der ersten Testszene.

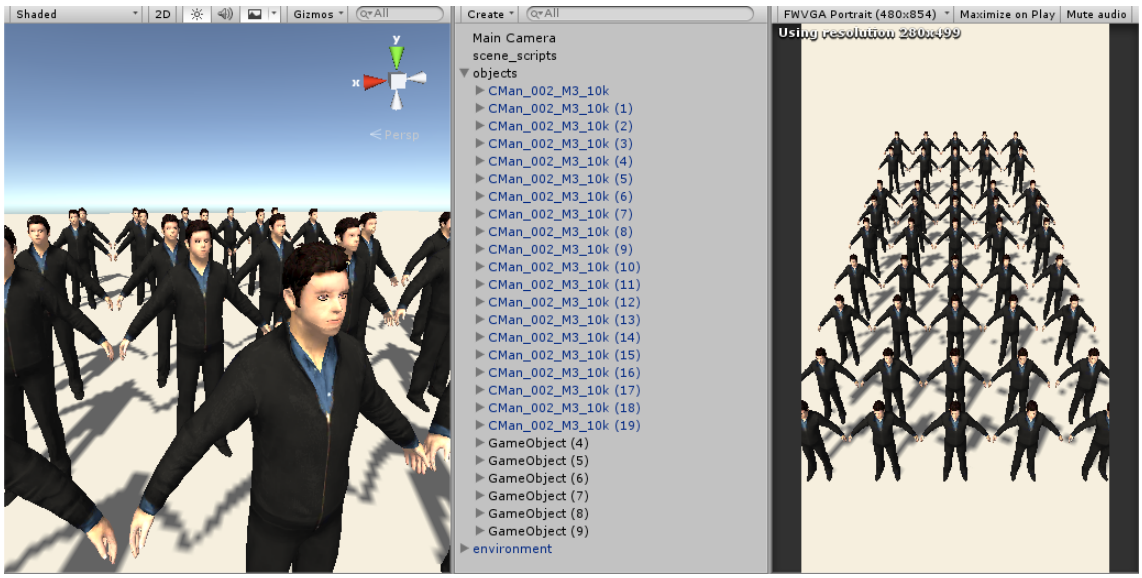

Abbildung 16: Testblock 2 mit 50 Charaktermodellen

4.3.3 Testblock 3

Der Aufbau dieses in Abbildung 17 gezeigten Testabschnitts entspricht dem von Testblock 1. Bei den Charakteren werden anstatt der *Legacy-Shader* die *PBR-Shader* von Unity 5 genutzt. Jedes Charaktermodell greift auf 3 *PBR-Shader* zu.

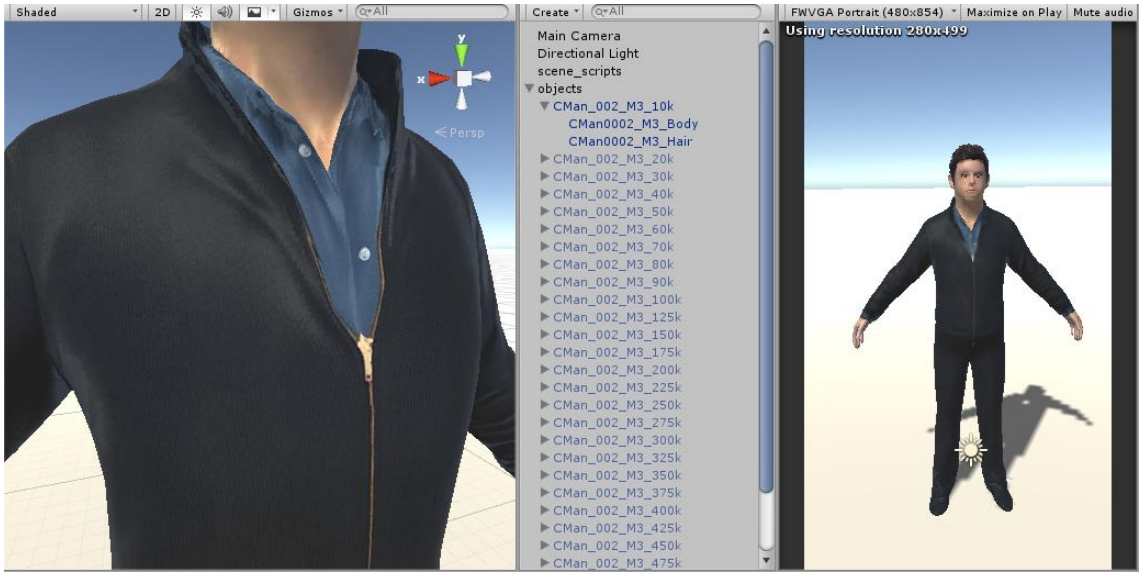

Abbildung 17: Testblock 3 mit PBR Shadern

4.3.4 Testblock 4

Der vierte Testabschnitt besteht aus 9 Charaktermodellen. Jeder der Charaktere besteht aus 10.000 *Triangles*. Die Anzahl der Modelle bleibt konstant. Die veränderliche Größe in diesem Testabschnitt ist die Anzahl der Lichtquellen. Diese sind Punktlichter, die gleichmäßig in alle Richtungen strahlen. Alle 15 Lichtquellen erzeugen dynamische Schatten. Abbildung 18 zeigt die Szene mit allen aktiven Lichtquellen.

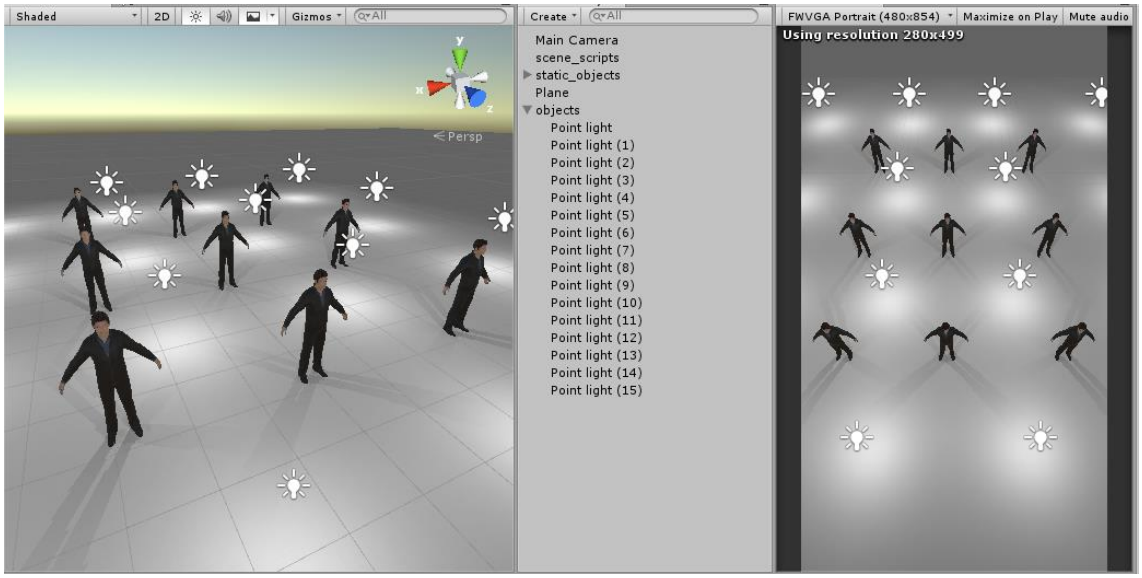

Abbildung 18: Punktlichter in Testblock 4

Die Szenendauer beträgt insgesamt 30 Sekunden. Aller zwei Sekunden wird eines der 15 Punktlichter aktiviert bis schließlich alle 15 die Szene beleuchten. Neben den Punktlichtern wird die Szene vom Umgebungslicht gleichmäßig aufgehellt.

4.3.5 Testblock 5

In diesem Abschnitt wird Variante des Charaktermodells genutzt, die über *Bones* animiert wird. Auf den *Bones* des Charakters liegt eine sich wiederholende Animation. Das Charaktermodell entspricht dem aus den vorhergehenden Tests. Es wird die Variante mit 10.000 *Triangles* verwendet. Das in Abbildung 19 gezeigte Modell besitzt 53 *Bones*. Der Aufbau von Testblock 5 entspricht der ersten Szene von Testblock 2.

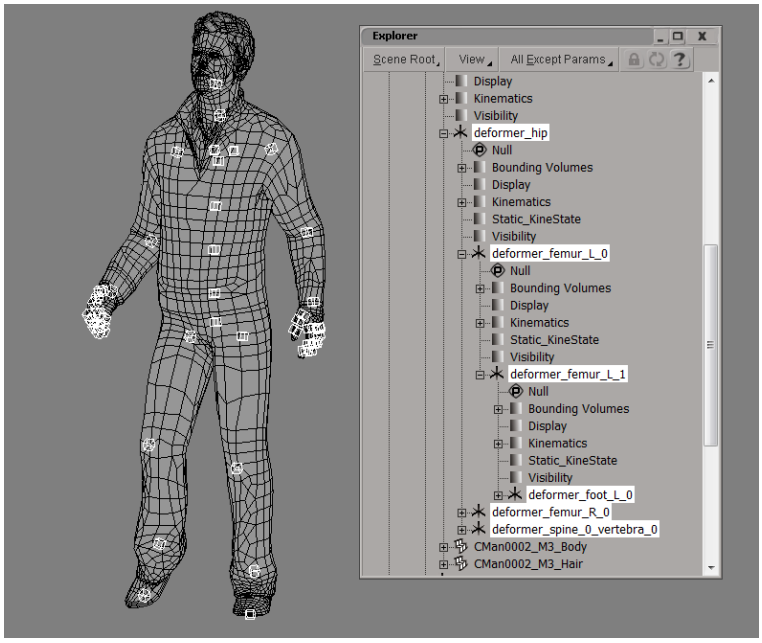

Abbildung 19: Charaktermodell mit Bones in Autodesk Softimage

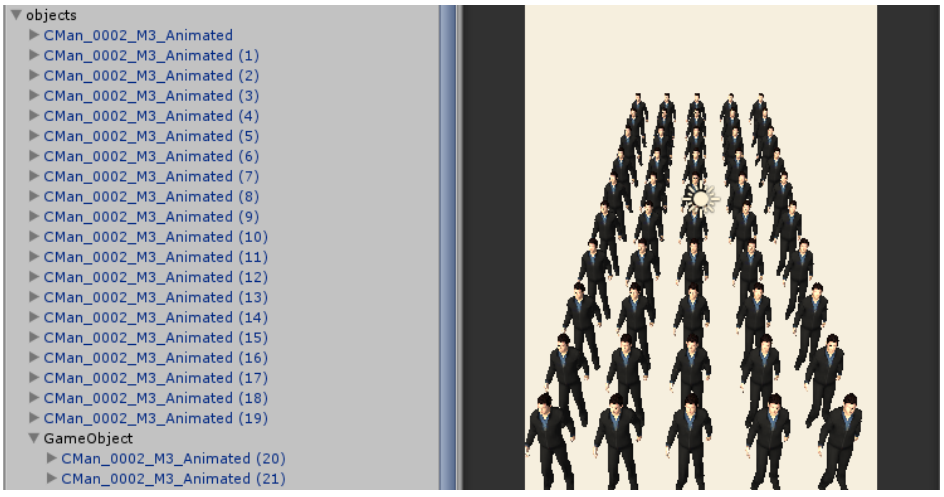

Abbildung 20: Animierte Charaktermodelle in Unity

5 Prüfung der Forschungsfragen

Die Leistungsanalyse-Anwendung aus Kapitel 4 besteht aus zwei Testanwendungen.[100][101] Mit den Anwendungen werden *Forward* und *Deferred Rendering* getestet. Diese zwei Anwendungen wurden auf verschiedenen Endgeräten getestet. Alle Forschungsfragen werden anhand der vorliegenden Daten der Testanwendung beantwortet. Die Ergebnisse lassen sich nicht auf andere Anwendungen übertragen.

5.1 Testgeräte

Es wurden folgende Testgeräte für die Leistungsanalyse verwendet.

Testgerät	*Hersteller*	*Typ*	*SoC*	*RAM*	*Android-Version*
1	Samsung	Galaxy S3 GT-I9305	Samsung Exynos 4412	1 GB	4.4.4
2	Google	Nexus 6	Qualcomm Snapdragon 805	3 GB	5.1.1
3	Samsung	Galaxy Note 4	Qualcomm Snapdragon 805	3 GB	5.1.1

Tabelle 6: Android Testgeräte

5.2 Gibt es Leistungsunterschiede zwischen einem Objekt mit hohem Detailgrad und vielen Objekten mit niedrigerem Detailgrad?

Um diesen Sachverhalt zu prüfen wurden *Logfile*s der Testgeräte anhand der *Triangle*-Zahlen verglichen. Es wird von den *Triangles* der geladenen 3D-Modelle ausgegangen, da die Anzahl der in den *Logfiles* angezeigten *Triangles* von den in der Szene vorhandenen abweicht. Auf diesen Sachverhalt wird in Kapitel 6.3 näher eingegangen.

[100] Anlage CD /apps/profiling_app_forward.apk
[101] Anlage CD /apps/profiling_app_deferred.apk

Hierfür werden die Charaktermodelle mit 100.000 und 200.000 *Triangles* aus Testblock 1 mit verschiedenen Zahlen gleicher Charaktermodelle mit geringerem Detailgrad aus Testblock 2 und 3 verglichen. [102] Beispielweise werden 10 Modelle mit 10.000 *Triangles* dem hoch aufgelösten Modell aus Testblock 1 gegenübergestellt.

5.2.1 Testgerät 1

100.000 Triangles

In Testblock 1 beträgt die *Frame Time* bei der Darstellung eines Objektes mit 100.000 *Triangles* 33,1 ms. Unter Verwendung von niedriger aufgelösten Geometrien ist die *Frame Time* geringer. Bei 10 Objekten mit je 10.000 *Triangles* beträgt sie 18,7 ms. Bei 5 Objekten mit je 20.000 *Triangles* liegt sie bei 16,7 ms und bei 2 Objekten mit je 50.000 *Triangles* bei ebenfalls 16,7 ms.

Bei der Darstellung eines Hochaufgelösten Objektes musste das *Android*-Gerät mehr *Triangles* berechnen als das angezeigte 3D-Modell des Charakters tatsächlich besaß. Bei einem Modell mit 100.000 *Triangles* wurden in der *Logfile* 203.810 *Triangles* aufgezeichnet. Bei den niedriger aufgelösten Objekten schwankt die Zahl der *Triangles* zwischen 102.729 und 110.522. Zwischen den drei Tests mit niedrig aufgelösten Objekten und einem hoch aufgelösten Objekt gibt es im Durchschnitt eine Abweichung von 98.189 *Triangles*.

Eine höhere Anzahl an niedriger aufgelösten Objekten hat ebenfalls eine negative Auswirkung auf die Anwendungsleistung. Diese ist jedoch weniger stark als bei einem hoch aufgelösten Objekt. Grund hierfür sind die *Draw Calls*. Das heißt die Berechnungsschritte die für eine 3D-Umgebung nötig sind. Pro Objekt in der 3D-Umgebung sind zusätzliche Berechnungsschritte möglich, was die Anwendungsleistung verringert. Für die Darstellung eines Objektes mit 100.000 *Triangles* sind in der Analyseanwendung 14 *Draw Calls* nötig. Bei 10 Objekten sind es 33. Die Anwendungsleistung ist jedoch besser, da weniger *Triangles* berechnet werden müssen. Unter Verwendung von wenigen, etwas höher aufgelösten, Objekten sinken Die *Draw Calls* auf 18 und 9. Die Anzahl der *Triangles* bleibt in diesen beiden Fällen annähernd gleich. Die *Frame Time* verringert sich bei gleichbleibender *Triangle*-Anzahl und sinkenden *Draw Calls*. Durch die verringerte

[102] Anlage CD /profiler_app/profiler_app_galaxy_s3_4_4_4_forward_converted.csv

Frame Time bei wenigen, etwas höher aufgelösten, Objekten läuft die Anwendung flüssiger.

200.000 *Triangles*

Unter Verwendung von 200.000 *Triangles* kommen ähnliche Ergebnisse zustande. Bei einem Objekt mit 200.000 *Triangles* beträgt die *Frame Time* 52,7 ms. In diesem Fall werden 403.810 *Triangles* berechnet. Bei 20 Objekten mit je 10.000 *Triangles* beträgt die *Frame Time* 34,9 ms. Die Anwendung läuft also flüssiger. Unter Verwendung 10 Objekten mit je 20.000 *Triangles* steigt die Leistung der Anwendung nochmals. Die *Frame Time* beträgt in diesem Fall 26,9 ms. Mit nur 4 Objekten zu je 50.000 *Triangles* beträgt sie 23,3 ms.

Der Sachverhalt mit den *Draw Calls* bestätigt sich bei diesem Test erneut. 20 Objekte mit je 10.000 *Triangles* werden langsamer berechnet als 4 Objekte mit je 50.000 *Triangles*. Jedoch sind die 20 Objekte in kürzerer Zeit berechnet als ein hoch aufgelöstes Objekt mit 200.000 *Triangles*.

5.2.2 Fazit

Bei der Darstellung von höher Aufgelösten 3D-Modellen muss Unity viel mehr *Triangles* berechnen als bei vielen niedriger aufgelösten. Auf den Genauen Sachverhalt wird in Kapitel 6.3 näher eingegangen. Es empfiehlt sich 3D-Umgebungen aus mehreren Objekten mittlerer Auflösung zu konstruieren. Diese liefern eine bessere Anwendungsleistung als ein hoch aufgelöstes Objekt. Von der Verwendung vieler Objekte mit niedriger Auflösung ist ebenfalls abzusehen, da hierbei die *Draw Calls* stark ansteigen. Dies wirkt sich negativ auf die Anwendungsleistung aus. Die Anwendungsleistung ist in den Tests mit vielen niedrig aufgelösten Objekten dennoch besser als mit einem hoch aufgelösten.

5.3 Steigt der Leistungsbedarf bei der Verwendung vieler detaillierter 3D-Modelle linear an?

Hierfür wurde der Testabschnitt 2 der Analyseanwendung ausgewertet. Die Auswertung erfolgt mit Testgerät 1 und Testgerät 2. Da das ältere Testgerät 1 nur mit *Forward Rendering* darstellen kann wird bei Testgerät 2 auch der Datensatz verwendet, welcher mittels *Forward Rendering* erstellt wurde.[103][104]

5.3.1 Testgerät 1

Testblock 2.1

Abbildung 21 zeigt den Verlauf von *Frame Time* und *Triangles* von Testszene 2.1. In der Darstellung ist ein linearer Anstieg der *Frame Time* bei Erhöhung der *Triangles* zu erkennen. Bis Sekunde 18 ist bei der Kurve der *Frame Time* keine Steigung zu erkennen, dies liegt an der minimal-möglichen *Frame Time* von 16,7 ms.

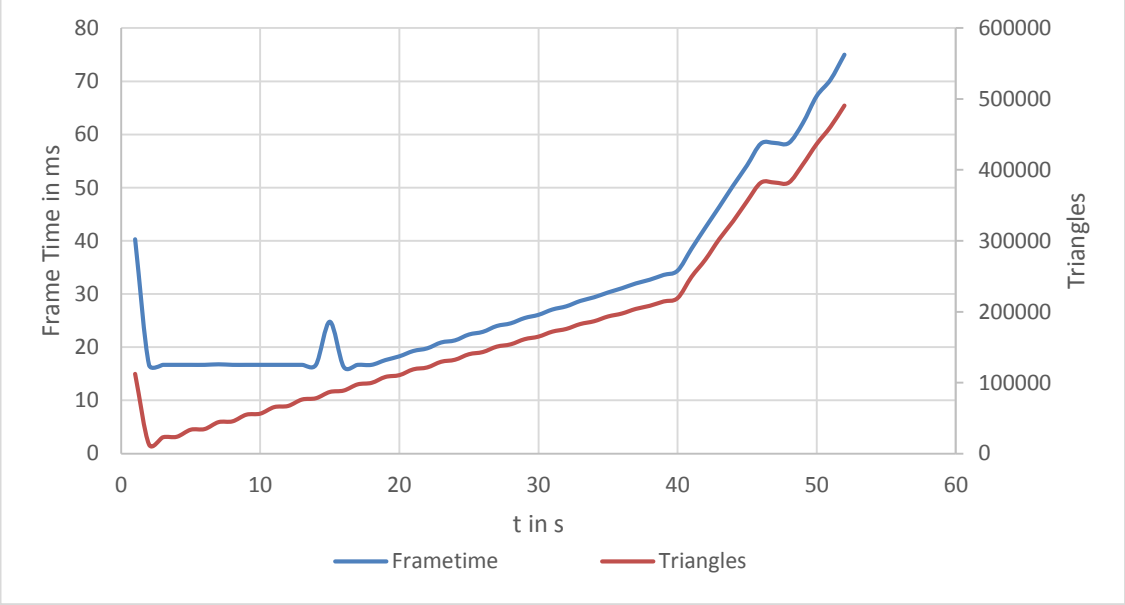

Abbildung 21: Frame Time und Triangle-Verlauf bei Testszene 2.1 auf Testgerät 1

[103] Anlage CD /profiler_app/profiler_app_galaxy_s3_4_4_4_forward_converted.csv
[104] Anlage CD /profiler_app/profiler_app_nexus_6_5_1_forward_converted.csv

Bis Sekunde 40 werden die 3D-Modelle der Charaktere, mit je 10.000 *Triangles*, einzeln eingeblendet. Ab Sekunde 40 werden die Objekte in Blöcken zu je fünf Modellen eingeblendet. Eine erste Abweichung von der *Frame Time* von 16,7 ms gibt es ab Sekunde 19. Zu diesem Zeitpunkt beträgt die *Frame Time* 17,6 ms. Im Bereich zwischen 19 und 40 Sekunden korrelieren *Frame Time* und *Triangle*-Anzahl stark miteinander. Der Korrelationskoeffizient beträgt $r = 0.9998$. Demzufolge gibt es, wie bereits im Diagramm zu sehen einen starken linearen Zusammenhang zwischen der *Triangle*-Anzahl und der *Frame Time*.

Auffällig ist der Bereich ab Sekunde 41 bis zum Ende der 52 Sekunden dauernden Szene. In diesem Bereich ist ein deutlicher Bruch im Verlauf der Kurve zu erkennen. Sowohl die Anzahl der *Triangles*, als auch die *Frame Time* brechen für 2 Sekunden deutlich ein. Der Zusammenhang zwischen beiden Kurven ist weiterhin linear und korreliert mit $r = 0.9997$. Grund für den Einbruch der *Frame Time* und *Triangles* für 2 Sekunden, ist ein Fehler in der Analyseanwendung. Die 5 Charaktermodelle die zu diesem Zeitpunkt geladen werden sollten wurden nicht eingeblendet. Diese waren in der *Unity*-Szene deaktiviert.

Testblock 2.3

Abbildung 22 zeigt Szene 2.3. In dieser Szene wurden Schrittweise Modelle mit je 30.000 *Triangles* geladen. Nach 9 *Sekunden* steigt die *Frame Time* in diesem Testszenario über den minimalen Wert von 16,7 ms. Die Linearität wird wie im vorhergehenden Test bis Sekunde 40 geprüft. Der Korrelationskoeffizient für die 31 Sekunden beträgt $r = 0.9905$. Demzufolge ist ebenfalls ein starker linearer Zusammenhang zu erkennen.

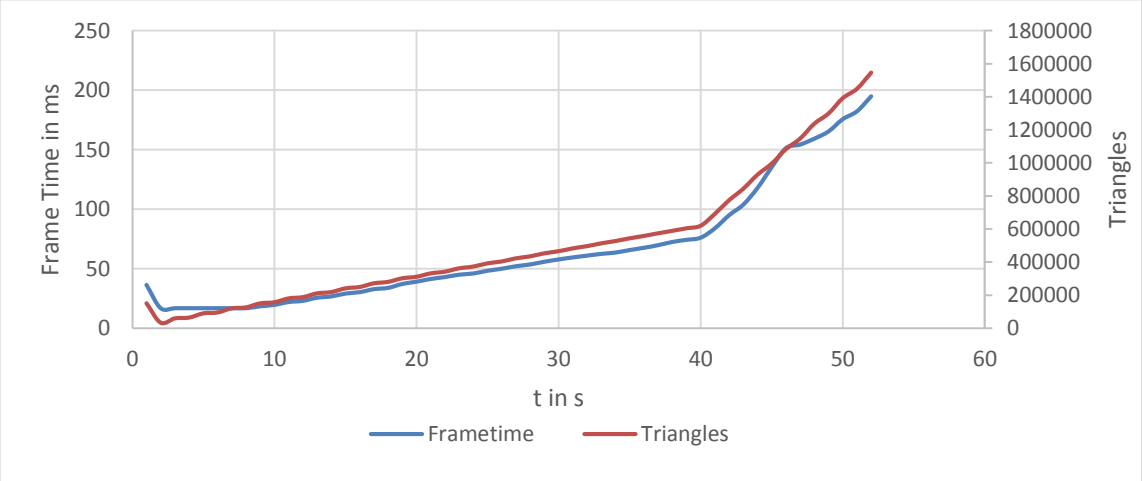

Abbildung 22: Frame Time- und Triangle-Verlauf bei Testszene 2.3 auf Testgerät 1

Wie auch in Testblock 2.1 ist im Zeitabschnitt zwischen Sekunde 41 und dem Ende der Szene ein Bruch zu erkennen. Im Gegensatz zu Testblock 2.1 wurden in diesem Abschnitt jedoch alle Charaktermodelle geladen. Dies ist deutlich anhand der linear verlaufenden *Triangle*-Kurve in Abbildung 22 zu erkennen. Die lineare Abhängigkeit von *Frame Time* und *Triangles* ist auch hier gegeben. Fällt jedoch mit $r = 0.9874$ geringer aus.

Testblock 2.5

In Testblock 2.5 bestätigen sich die linearen Zusammenhänge zwischen dem Detailgrad der Objekte und der *Frame Time* erneut. In diesem Testblock werden Charaktermodelle zu je 50.000 *Triangles* geladen. Abbildung 35 zeigt die Verläufe von *Frame Time* und *Triangles* bei Testblock 2.5. Die lineare Abhängigkeit ist in beiden Teilen des Testblocks 2.5 gegeben. Sowohl im ersten Abschnitt, in dem die Modelle einzeln geladen, als auch im zweiten korrelieren beide Kurven positiv-linear miteinander. Ab Sekunde 5 wird die *Frame Time* von 16,7 ms erstmals überschritten. In den 35 Sekunden bis die Objekte in Blöcken geladen werden beträgt $r = 0.9997$. Im zweiten Abschnitt des Testblocks beträgt $r = 0.9979$.

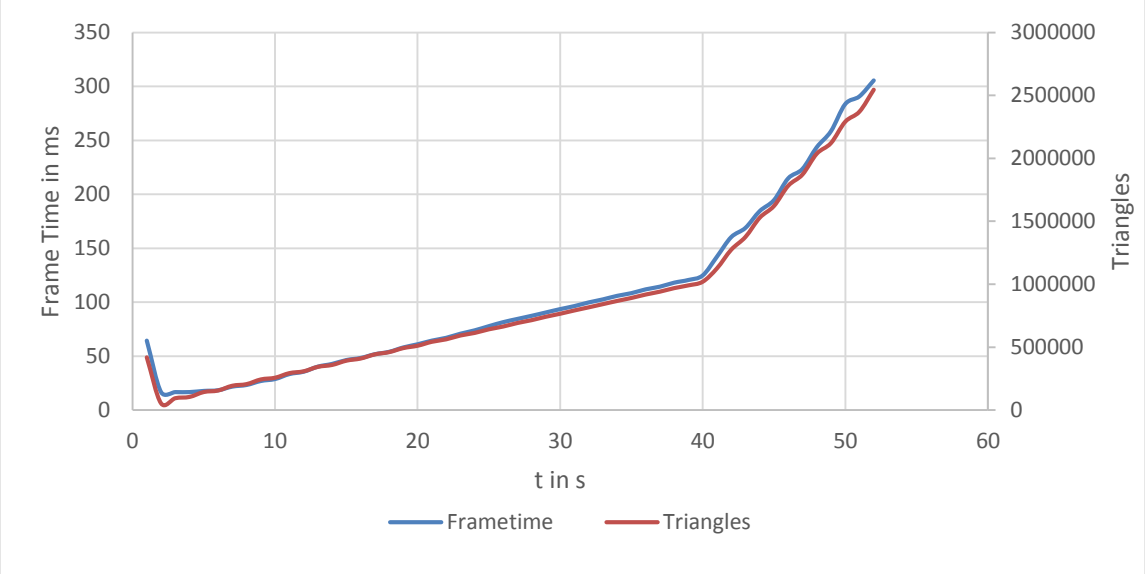

Abbildung 23: Frame Time- und Triangle-Verlauf bei Testszene 2.5 auf Testgerät 1

5.3.2 Testgerät 2

Testblock 2.1

Im ersten Abschnitt von Testblock 2 weicht die *Frame Time* bei Testgerät 2 nur minimal von 16,7 ms ab. Abbildung 24 zeigt diesen Sachverhalt. Bis Sekunde 45 gibt es keine Abweichung von der minimal möglichen *Frame Time*. Anschließend unterliegt die *Frame Time* Schwankungen von 2,28 ms. Die *Frame Time* sinkt nach kurzen Sprüngen jedoch wieder auf 16,7 ms.

Aufgrund der nicht steigenden *Frame Time* existiert kein linearer Zusammenhang in beiden Teilbereichen des Testblocks 2.1 auf Testgerät 2. Der Korrelationskoeffizient im ersten Bereich beträgt $r = -2{,}144$. Im zweiten Bereich beträgt $r = 0{,}0548$.

Ohne Berücksichtigung der minimalen *Frame Time* von 16,7 ms bedeutet dies, dass es bei Testgerät 2 in Testblock 2.1 keinen linearen Zusammenhang zwischen der *Frame Time* und den *Triangles* gibt.

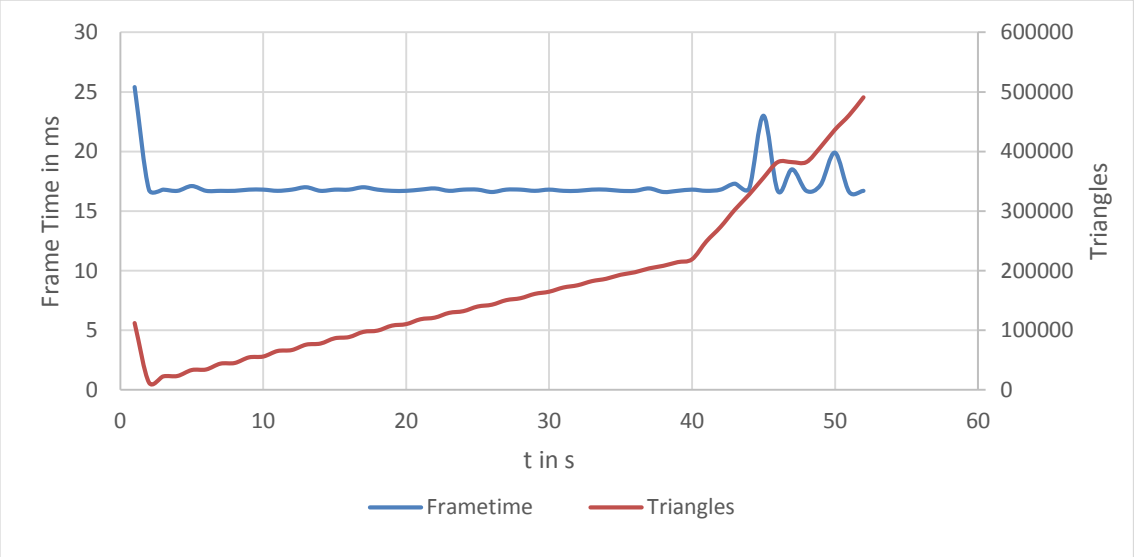

Abbildung 24: Frame Time- und Triangle-Verlauf bei Testszene 2.1auf Testgerät 2

Testblock 2.3

Abbildung 25 zeigt die Kurven von *Frame Time* und *Triangles* bei Testblock 2.3 unter Testgerät 2. Anhand des Diagramms ist zu erkennen, dass es bis Sekunde 40 mit $r = 0{,}0190$ keinen Zusammenhang zwischen *Frame Time* und *Triangles* gibt. Wie bei

Testgerät 1 ist der Zusammenhang bei vielen dargestellten *Triangles* mit $r = 0{,}9988$ positiv linear.

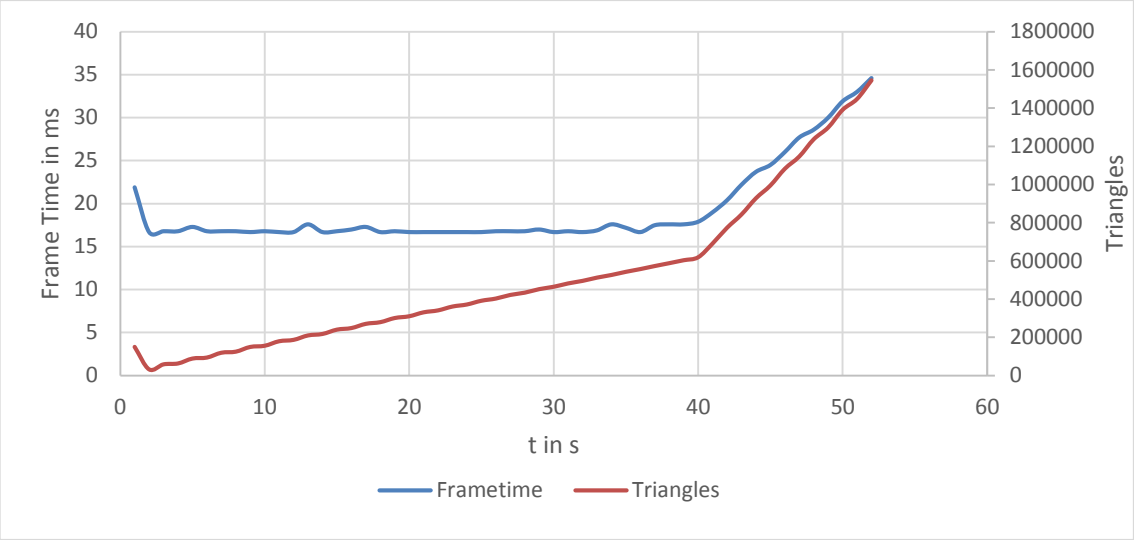

Abbildung 25: Frame Time- und Triangle-Verlauf bei Testszene 2.3 auf Testgerät 2

Testblock 2.5

Abbildung 26 zeigt, dass es bei Testgerät 2 ebenfalls einen linearen Zusammenhang zwischen dem Detailgrad der Objekte und der Leistung der Anwendung gibt. Sowohl zwischen Sekunde 23 und 40 mit $r = 0{,}9948$, als auch zwischen 41 und dem Ende des Testblocks gibt es mit $r = 0{,}9997$ eindeutig lineare Zusammenhänge zwischen *Frame Time* und *Triangles*.

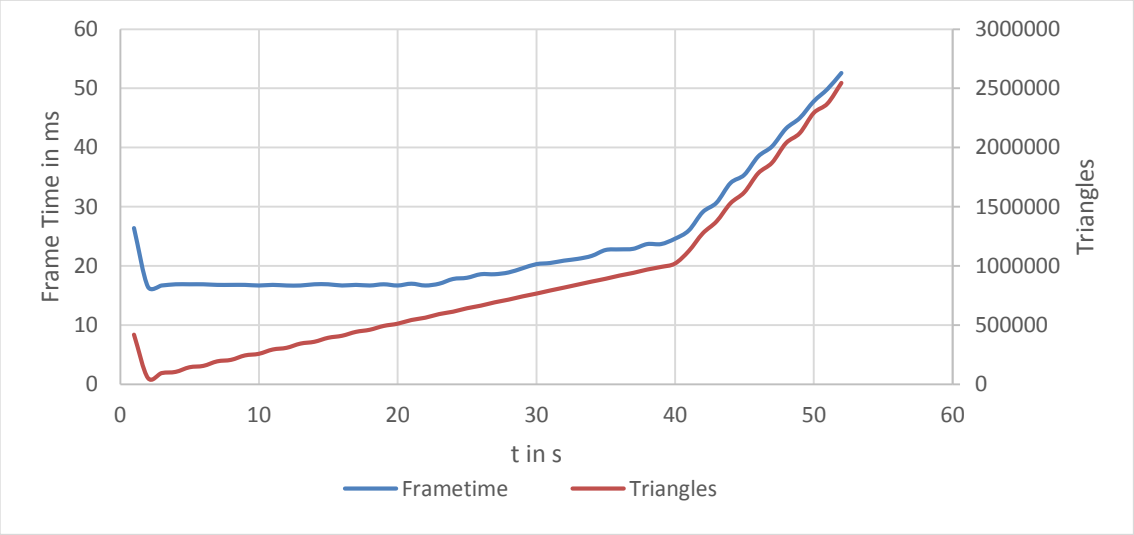

Abbildung 26: Frame Time- und Triangle-Verlauf bei Testszene 2.5 auf Testgerät 2

5.3.3 Fazit

Die Analyse der *Profiler-Logs* der Testgeräte 1 und 2 ergab, dass es eindeutig lineare Zusammenhänge zwischen der Verwendung vieler detaillierter 3D-Modelle und der *Frame Time* gibt. Die linearen-Abhängigkeiten konnten jedoch in Bereichen, in denen die *Frame Time* unter 16,7 ms liegt nicht bestimmt werden, da niedrigere Werte aufgrund der maximalen Bildrate von 60 fps bei *Android*-Geräten nicht möglich sind. In diesen Abschnitten der *Logfiles* sind die Kurven anhand der aufgezeichneten Daten nicht linear voneinander abhängig.

5.4 Wie viele Lichtquellen können auf einem spezifischen Android-Gerät parallel aktiv sein, um eine flüssige Bildrate zu ermöglichen?

Um diesen Sachverhalt zu prüfen, werden die mit Testgerät 1 und 3 aufgezeichneten Daten analysiert. Es werden die drei verschiedenen Beleuchtungsszenarien betrachtet. Es ist zu erwarten, dass ein modernes *Android*-Gerät besser mit vielen Lichtquellen skaliert.

5.4.1 Testgerät 1

Da bei Testgerät 1 nur *Forward Rendering* analysiert werden kann wird auf diesem Gerät nur dieser *Renderpfad* geprüft.[105]

Forward Rendering

Die Tests mit verschieden vielen Lichtquellen brachten, unabhängig der Schatteneinstellung der Lichtquellen, ähnliche Ergebnisse. Im Durchschnitt weichen die *Frame Time* Werte aller drei Durchläufe nur um 1,118 ms voneinander ab. Abbildung 27 verdeutlicht diesen Sachverhalt.

[105] Anlage CD /profiler_app/profiler_app_galaxy_s3_4_4_4_forward_converted.csv

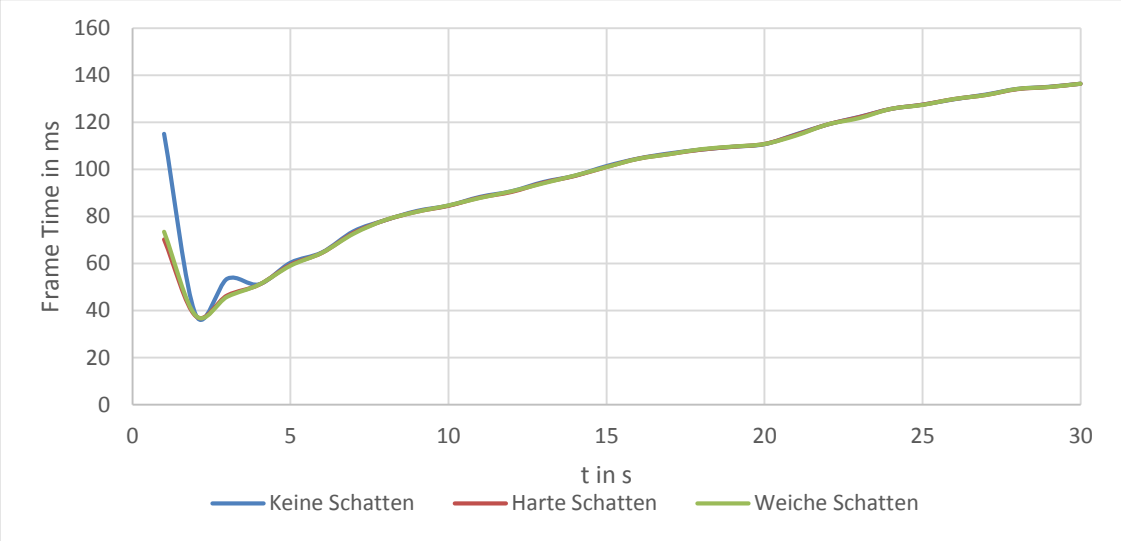

Abbildung 27: Frame Time Verlauf verschiedener Lichtquellen unter Testgerät 1

Während die Szene auf dem *Android*-Gerät ausgeführt wurde, wurden ebenfalls keine Schatten dargestellt. Dieser Sachverhalt wird in Kapitel 6.4 genauer analysiert.

Um die Forschungsfrage zu klären wurde überprüft, wann eine *PAL*-Bildrate von 25 fps unterschritten wird. Für die *Frame Time* gilt hierbei, dass 40 ms nicht unterschritten werden dürfen. Im Fall von Testgerät 1 tritt dies bereits ab Sekunde 3 auf. In Sekunde 2 der Anwendung beträgt die *Frame Time* noch 38,2 ms. Jeder darauffolgende Wert liegt oberhalb des kritischen Wertes von 40 ms. Zu diesem Zeitpunkt wird eine dynamische Lichtquelle dargestellt.

5.4.2 Testgerät 3

Bei Testgerät 3 ist zu erwarten, dass die kritische *Frame Time* für die *PAL*-Bildrate erst bei einer größeren Anzahl von Lichtquellen erreicht wird. Das Gerät wurde mit *Forward* und *Deferred Rendering* analysiert. [106][107]

Forward Rendering

In Abbildung 28 wird deutlich, dass die Einstellung der Schatten ebenfalls keine Auswirkung auf die Anwendungsleistung hat. Wie auch bei Testgerät 1 wurden auf Testgerät 3

[106] Anlage CD /profiler_app/profiler_app_note_4_5_1_forward_converted.csv
[107] Anlage CD /profiler_app/profiler_app_note_4_5_1_deferred_converted.csv

keine Schatten angezeigt. Die Ergebnisse wichen durchschnittlich um 0,511 ms voneinander ab. Diese Abweichung ist auf Messungenauigkeiten zurückzuführen.

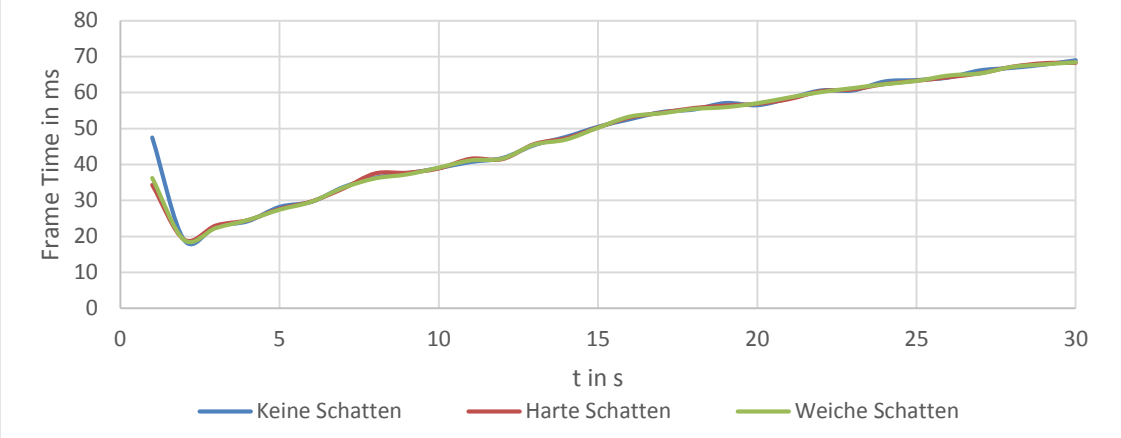

Abbildung 28: Frame Time Verlauf verschiedener Lichtquellen unter Testgerät 3 mit Forward Rendering

Wie erwartet ermöglicht das modernere *Android*-Gerät die Darstellung von mehr Lichtquellen bis die Bildrate zu niedrig wird. Im Gegensatz zu Testgerät 1 können fünf Lichtquellen dargestellt werden bis die *Frame Time* über 40 ms steigt.

Deferred Rendering

Unter *Deferred Rendering* gibt es deutliche Unterschiede zwischen der Darstellung mit und ohne Schatten. Die Durchschnittliche Abweichung der Messwerte voneinander beträgt 26,105 ms. Anhand der Verläufe in Abbildung 29 ist jedoch zu erkennen, dass es keine Unterschiede zwischen harten und weichen Schatten gibt. Die Unterschiede zwischen harten und weichen Schatten betragen im Durchschnitt 1,131 ms.

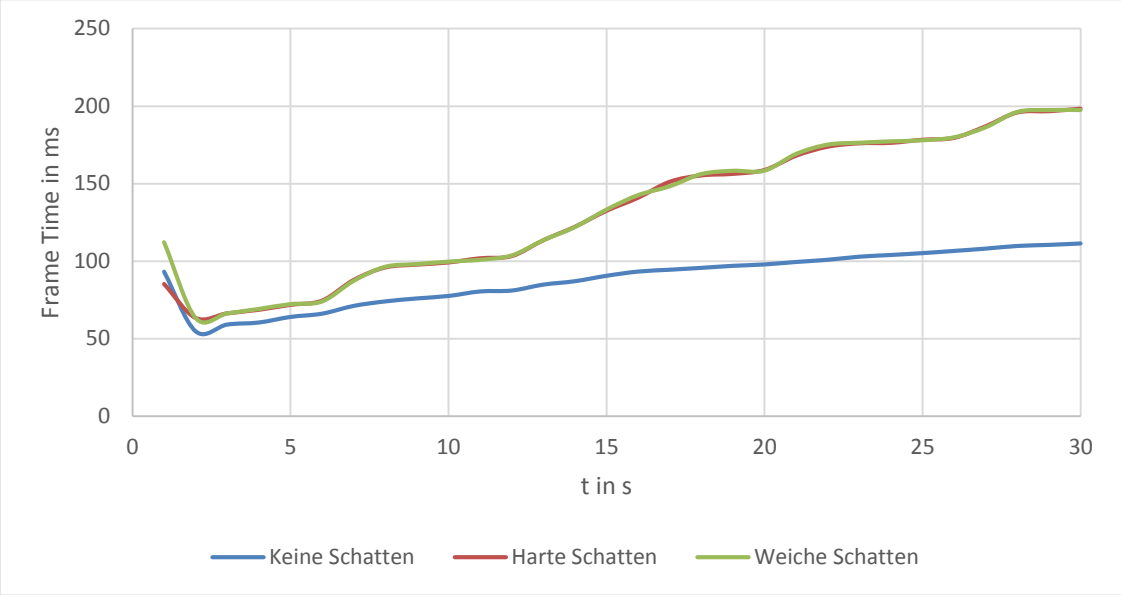

Abbildung 29: Frame Time Verlauf verschiedener Lichtquellen unter Testgerät 3 mit Deferred Rendering

Grund für die gleichen Verläufe der Messwerte für harte und weiche Schatten ist, dass es mit Unity derzeit nicht möglich ist weiche Schatten auf mobilen Endgeräten zu berechnen. Nachfolgender Auszug aus der Unity-Dokumentation erklärt diesen Sachverhalt:

> *„Shadows on Android and iOS have these limitations: soft shadows are not available(…)“.*[108]

Unter Verwendung von *Deferred Rendering* ist auf Testgerät 2 zu keinem Zeitpunkt eine flüssige Bildrate möglich. Bereits zu Beginn der Datenreihe liegt die *Frame Time* über dem für die *PAL*-Bildrate kritischen Wert.

Ohne dargestellte Schatten beträgt die *Frame Time* mit einer Lichtquelle bereits 54,7 ms. Mit aktivierten Schatten bei einer Lichtquelle erhöht sich der Wert auf 63,4 ms.

Bei 15 Dargestellten Lichtquellen gibt es, ohne Schatten, gegenüber *Forward Rendering* nahezu eine Verdoppelung der *Frame Time*. Bei *Forward Rendering* und 15 Lichtquellen beträgt diese 67,8 ms. Mit *Deferred Rendering* steigt der Wert auf 110,4 ms.

[108] UNITY, http://docs.unity3d.com/Manual/TroubleShooting.html, 20.10.15

5.4.3 Fazit

Mit Unity ist es derzeit nicht möglich weiche Schatten auf mobilen Endgeräten darzustellen. Anhand der Tests wurde deutlich, dass Schatten mehrerer Lichtquellen nur unter *Deferred Rendering* dargestellt wurden. Ältere Endgeräte, welche diesen *Renderpfad* nicht beherrschen, können jedoch keine Schatten bei Punktlichtern darstellen. In Kapitel 0 wird näher auf diesen Sachverhalt eingegangen. Unter *Deferred Rendering* ist, sowohl mit als auch ohne Schatten, die Berechnung von Lichtquellen sehr langsam

5.5 Wie verhalten sich Android-Geräte verschiedener Generationen mit vielen sichtbaren animierten Charakteren in einer Szene?

Für diesen Test wurde das Testgerät 1 mit dem modernen Testgerät 2 verglichen.[109] [110] Es ist zu erwarten, dass es deutliche Unterschiede zwischen beiden Geräten gibt.

5.5.1 Testgerät 1

In Abbildung 30 ist der Verlauf der Kurven für *Frame Time* und Charakteranzahl für Testblock 5 zu sehen. Bei der Analyseanwendung können bis zu 15 Charaktere gleichzeitig dargestellt werden ohne, dass die kritische *Frame Time* für die *PAL*-Bildfrequenz überschritten wird.

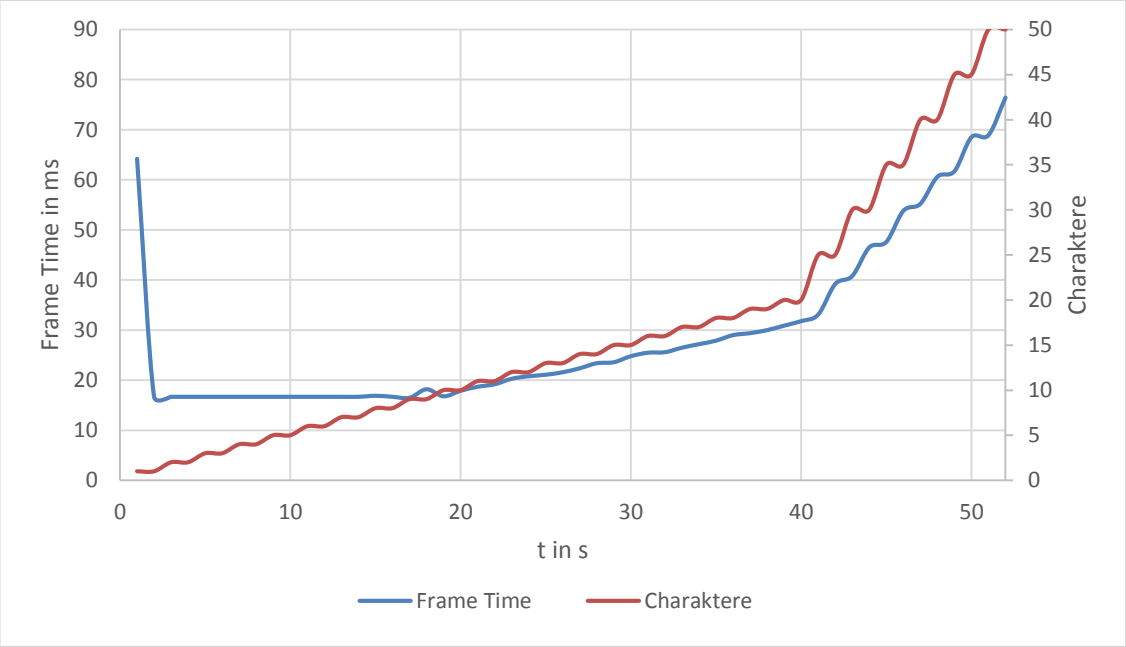

Abbildung 30: Darstellung der Frame Time und Charakteranzahl bei Testgerät 1

[109] Anlage CD /profiler_app/profiler_app_galaxy_s3_4_4_4_forward_converted.csv
[110] Anlage CD /profiler_app/profiler_app_nexus_6_5_1_forward_converted.csv

5.5.2 Testgerät 3

Das moderne Testgerät 3 ist in der Lage mehr Charaktere darzustellen. Während des gesamten Tests stieg die *Frame Time* nicht über den kritischen Wert von 40 ms. Die Daten von Testgerät 3 sind in Abbildung 30 visualisiert.

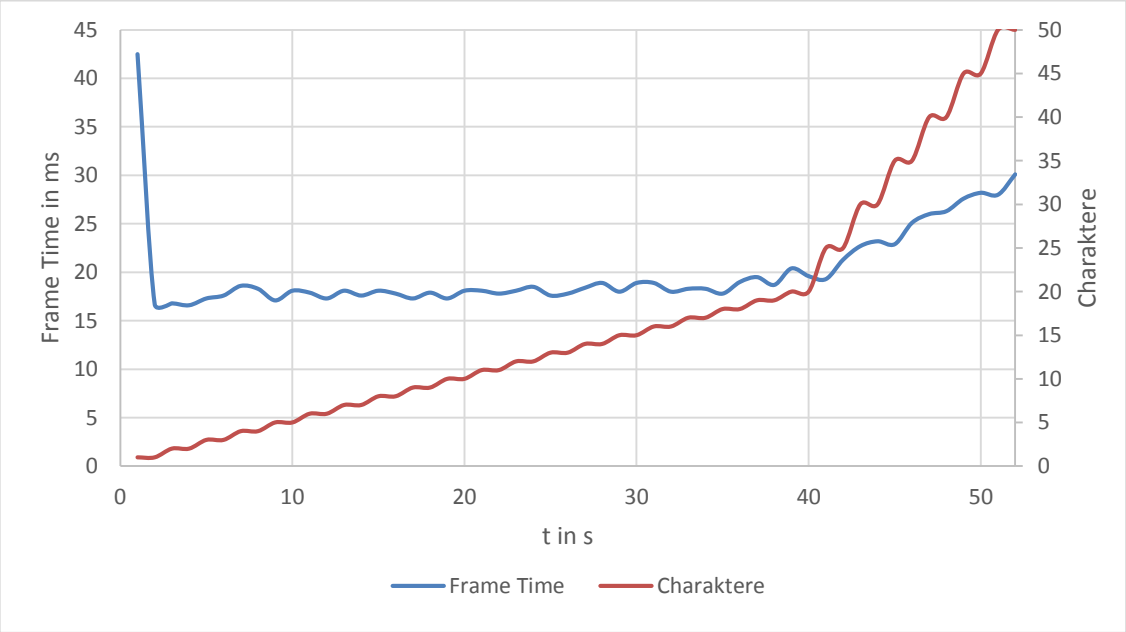

Abbildung 31: Darstellung der Frame Time und Charakteranzahl bei Testgerät 3

5.5.3 Fazit

Moderne mobile Endgeräte sind in der Lage viele animierte Charaktere gleichzeitig darzustellen. Bereits das 3 Jahre alte Testgerät 1 kann im vorliegenden Testszenario bis zu 15 Charaktere, mit je 10.000 *Triangles*, flüssig darstellen. Anhand der ausgewerteten Daten folgt, dass ein modernes *Android*-Gerät mit der Analyseanwendung bei der Darstellung von animierten-Charakteren nicht an seine Leistungsgrenze gebracht werden kann.

5.6 Ist der Leistungsunterschied zwischen Legacy- und PBR Shadern auf einem modernen Android Gerät geringer?

Für diesen Test werden Testgerät 1 und 2 miteinander verglichen. [111][112][113] Es wird geprüft, wie stark sich die *PBR-Shader* auf die Anwendungsleistung auswirken. Der Vorabtest der Testanwendung hat gezeigt, dass diese auf mobilen Endgeräten lauffähig sind. Es ist zu erwarten, dass die *PBR-Shader* höhere Anforderungen an die Hardware stellen. Dadurch wird die *Frame Time* voraussichtlich ansteigen.

5.6.1 Testgerät 1

Abbildung 32 zeigt die *Frame Time* der *Legacy-* und *PBR-Shader* bei Testgerät 1. Es ist deutlich zu erkennen, dass die *PBR-Shader* mehr Rechenzeit benötigen. Im Durchschnitt steigt die *Frame Time* mit *PBR-Shadern* um 18,13 ms.

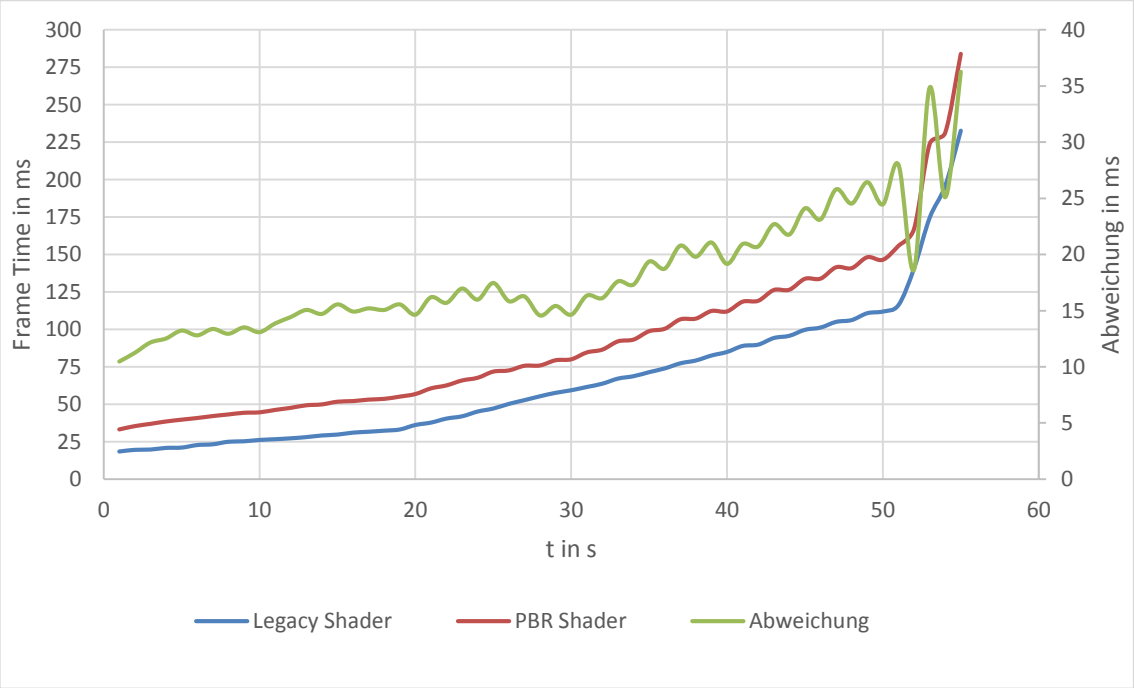

Abbildung 32: Frame Time Verläufe der Legacy- und PBR-Shader bei Testgerät 1

[111] Anlage CD /profiler_app/profiler_app_galaxy_s3_4_4_4_forward_converted.csv
[112] Anlage CD /profiler_app/profiler_app_nexus_6_5_1_forward_converted.csv
[113] Anlage CD /profiler_app/profiler_app_nexus_6_5_1_deferred_converted.csv

Die minimale Steigerung der *Frame Time* durch die *PBR Shader* beträgt 10,47 ms und die maximale 36,27 ms. Anhand der Abweichungskurve in Abbildung 32 ist deutlich zu erkennen, dass der Unterschied zwischen *Legacy*- und *PBR-Shadern* bei steigender Szenenkomplexität größer wird.

Ausgehend von einer flüssigen *PAL*-Bildrate von 25 fps beträgt die kritische *Frame Time* 40 ms. Unterhalb dieser *Frame Time* lassen sich Ruckler bei der Echtzeitdarstellung erkennen. Durch die Verwendung der *PBR-Shader* wird dieser kritische Wert bereits bei geringerer Szenenkomplexität erreicht. Bereits nach 7 Sekunden und bei 63.810 gerenderten *Triangles* sinkt die Bildrate unter 25 *Frames* pro Sekunde. Unter Verwendung der *Legacy-Shader* wird eine *Frame Time* von 40 ms erst nach 23 Sekunden und bei 283.752 gerenderten *Triangles* erreicht.

Aufgrund der fehlenden *OpenGL ES 3.0 & MRT* Kompatibilität von Testgerät 1 ist kein Vergleich zwischen *Forward* und *Deferred Shading* möglich. Beim Ausführen der *profiler_app_deferred.*apk wurde automatisch auf *Forward Rendering* gewechselt. Die Abweichungen zwischen den beiden Testanwendungen liegen innerhalb der in Kapitel 0 ermittelten Standardabweichung.

5.6.2 Testgerät 2

Bei aktuelleren *Android*-Smartphones als dem Testgerät 1 ist zu erwarten, dass der Unterschied zwischen *Legacy*- und *PBR-Shadern* geringer ausfällt. Dies wird im folgenden Abschnitt geprüft. Hierbei wird das Galaxy S3 von 2012 mit dem Nexus 6 von 2015 Gegenübergestellt.

Forward Rendering

Unter Verwendung von *Forward Rendering* ist die Abweichung zwischen den *Legacy* und *PBR-Shadern* signifikant kleiner als bei Testgerät 1.[114] Im Mittel weichen die Werte um 3,82 ms voneinander ab. Die maximale Abweichung beträgt 16,19 ms, die minimale beträgt 0,07 ms. Bei dem drei Jahre älteren Galaxy S3 fällt der Unterschied zwischen *Legacy*- und *PBR-Shadern* wesentlich höher aus als bei einem aktuellen *Android*-Gerät. Abbildung 33 zeigt die *Frame Time*-Verläufe der *Legacy*- und *PBR-Shader* bei Testgerät

[114] Vgl. Anlage CD Logfiles/profiler_app_nexus_6_5_1_forward_converted.csv

2. Eine steigende Szenenkomplexität führt wie auch bei Testgerät 1 zu einer höheren Abweichung. Die Abweichung fällt wesentlich geringer aus als bei Testgerät 1.

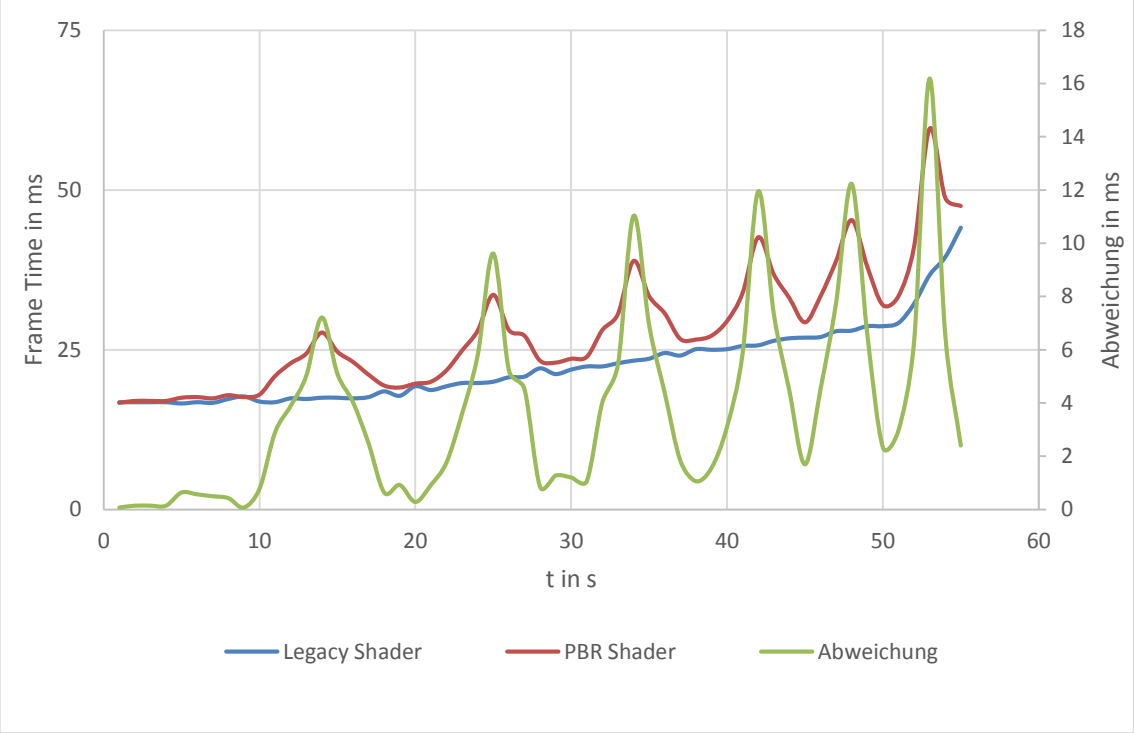

Abbildung 33: Frame Time Verläufe der Legacy- und PBR-Shader bei Testgerät 2 mit Forward Rendering

Eine flüssige *Frame Rate* von 25 fps wird mit den *Legacy-Shadern* erst nach 55 Sekunden unterschritten. Die Anzahl der dargestellten *Triangles* beträgt hierbei 1.703.781. Bei Verwendung der *PBR-Shader* schwankt die *Frame Time* stärker. Es gibt bereits bei 43 Sekunden eine erste Überschreitung der *Frame Time* von 40 ms. Diese sinkt im Anschluss jedoch wieder. Diese kurzen *Frame Time*-Anstiege sind auf die thermische Belastung des Prozessors zurückzuführen.

Aus im folgenden Absatz gezeigten *Logfile profiler_app_nexus_6_5_1_forward.txt* ist zu erkennen, dass nach einer Verringerung des CPU-Taktes die *Frame Time* ansteigt.[115] Nachdem die thermische Belastung der CPU wieder gesunken ist, wird der Takt angehoben, was in einer geringeren *Frame Time* resultiert.

[115] Vgl. Anlage CD Logfiles/profiler_app_nexus_6_5_1_forward.txt

profiler_app_nexus_6_5_1_forward.txt

```
09-14 14:54:00.617: I/ThermalEngine(1558): Mitigation:GPU[0]:389000000 Hz
(…)
09-14 14:54:01.492: D/Unity(19855): frametime>     min: 10.4   max: 35.2   avg: 24.4
(…)
09-14 14:54:03.153: D/Unity(19855): frametime>     min:  5.2   max: 45.6   avg: 27.7
(…)
09-14 14:54:03.628: I/ThermalEngine(1558): Mitigation:GPU[0]:500000000 Hz
(…)
09-14 14:54:04.627: D/Unity(19855): frametime>     min:  7.3   max: 38.4   avg: 24.7
(…)
09-14 14:54:06.010: D/Unity(19855): frametime>     min: 11.0   max: 35.6   avg: 23.1
(…)
09-14 14:54:06.639: I/ThermalEngine(1558): Mitigation:GPU[0]:600000000 Hz
(…)
09-14 14:54:07.269: D/Unity(19855): frametime>     min:  5.5   max: 37.4   avg: 21.1
```

Eine dauerhafte Überschreitung von 40 ms findet erst bei Sekunde 53 statt. Daraus folgt, dass unter Verwendung von *Forward Rendering* bei einem aktuellen *Android*-Gerät, die Unterschiede zwischen *Legacy*- und *PBR-Shadern* geringer ausgeprägt sind.

Die Überschreitungen der kritischen *Frame Time* von 40 ms unter Verwendung von *Forward Rendering* treten bei Testgerät 2 aufgrund von hohen Systemtemperaturen auf. Dadurch wird die CPU des Gerätes gedrosselt, was in einer geringeren Anwendungsleistung resultiert. Daraus folgt, dass hohe *Triangle*-Zahlen mit *PBR-Shadern* nur für eine kurze Zeit flüssig dargestellt werden können. Unter Verwendung der *Legacy-Shader* kam es zu keiner thermisch bedingten Drosselung. Die erste in der *Logfile* verzeichnete Drosselung trat nach dem Start von Testblock 3 ein.

profiler_app_nexus_6_5_1_forward.txt

```
09-14 14:53:39.896: I/Unity(19855): changing level to: testblock_03_01
(…)
09-14 14:53:57.600: I/ThermalEngine(1558): Mitigation:GPU[0]:500000000 Hz
```

Deferred Rendering

Unter Verwendung von *Deferred Rendering* stieg die *Frame Time* bei Testgerät 2 massiv an.[116] Abbildung 34 zeigt dieses Verhalten. Die Kurven von *Legacy*- und *PBR-Shadern* sind unter *Deferred Rendering* annähernd gleich. Die Abweichung beider Kurven beträgt im Mittel 1,31 ms. Die maximale Abweichung beträgt 13,15 ms und die minimale 0 ms.

[116] Vgl. Anlage CD profiler_app/profiler_app_nexus_6_5_1_forward_converted.csv

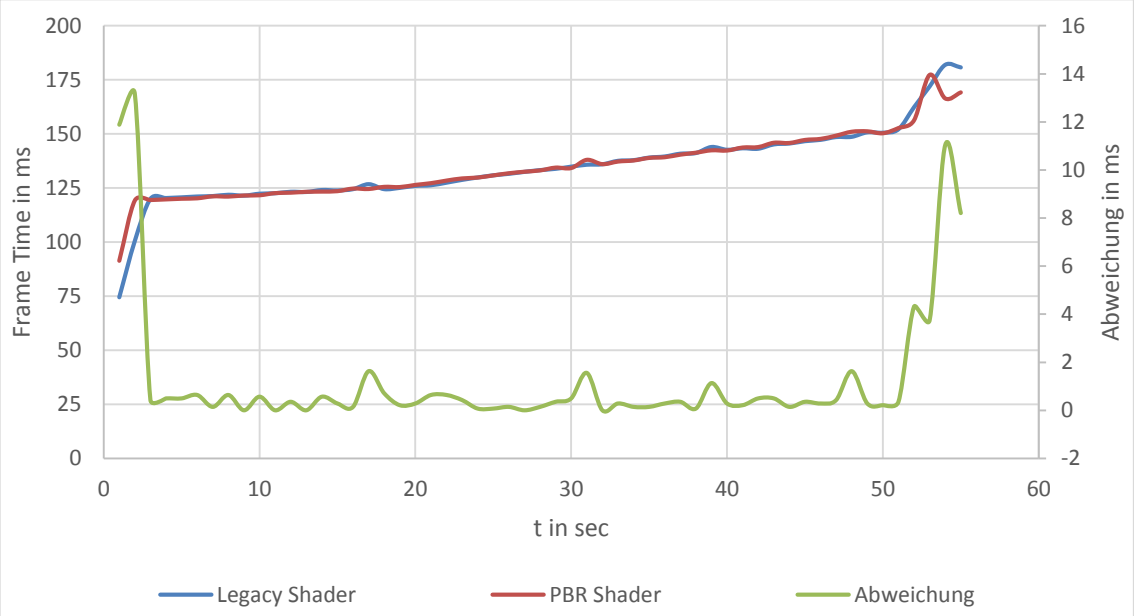

Abbildung 34: Frame Time Verläufe der Legacy- und PBR-Shader bei Testgerät 2 mit Deferred Rendering

Im Gegensatz zu *Forward Rendering* greift die thermische Drosselung der CPU bei *Deferred Rendering* in jedem Testabschnitt. Der folgende *Logfile*-Auszug[117] zeigt die CPU-Drosselung wenige Sekunden nach dem Start von Testblock 1.

profiler_app_nexus_6_5_1_deferred.txt

09-14 15:14:50.819: I/Unity(20932): CMan_002_M3_10k (UnityEngine.GameObject)
(…)
09-14 15:14:55.336: D/Unity(20932): frametime> min: 0.0 max: 106.6 avg: 69.8
(…)
09-14 15:14:57.870: I/ThermalEngine(1558): Mitigation:GPU[0]:500000000 Hz
(…
09-14 15:14:59.795: D/Unity(20932): frametime> min: 46.6 max: 95.5 avg: 74.5
(…)
09-14 15:15:00.887: I/ThermalEngine(1558): Mitigation:GPU[0]:389000000 Hz

Bei Testgerät 2 ist die Anwendungsleistung unter *Deferred Rendering* signifikant schlechter. Im Gegensatz zu *Forward Rendering* liegt die durchschnittliche Steigerung der *Frame Time* mit den *Legacy-Shadern* bei 79,32 ms. Mit den *PBR-Shadern* beträgt sie 75,60 ms.

[117] Vgl. Anlage CD profiler_app/profiler_app_nexus_6_5_1_forward.txt

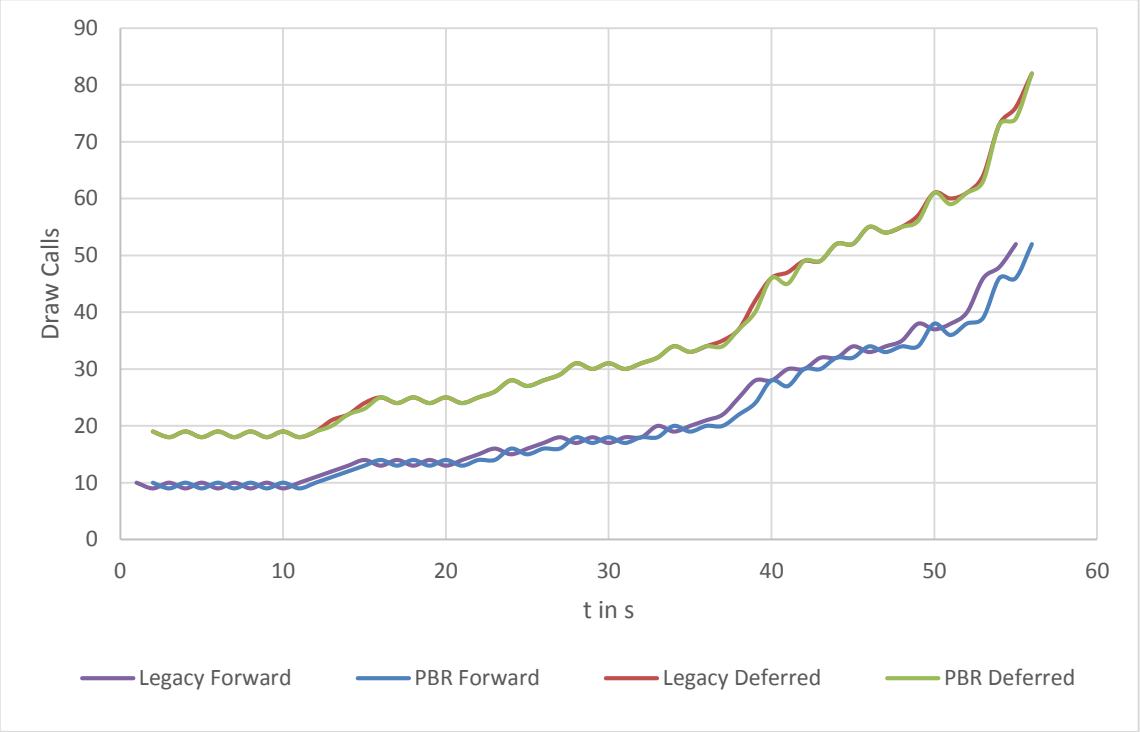

Abbildung 35: Draw Calls von Forward und Deferred Rendering bei Testgerät 2

Abbildung 35 zeigt, dass die höhere *Frame Time* auf eine gestiegene *Draw Call* Anzahl zurückzuführen ist. Neben den *Draw Calls* sind unter *Deferred Rendering* auch die cpu-*player*- und *render*-Zeiten des *player-detail* Blockes erhöht. Dies zeigt, dass dieser *Renderpfad* die Hardware wesentlich stärker belastet als *Forward Rendering*

5.6.3 Fazit

Auf einem modernen *Android*-Gerät fällt der Leistungsunterschied zwischen den *Legacy*- und *PBR-Shadern* geringer aus. Bei Testgerät 2 kam es jedoch bei der Ausführung der *PBR*-Shader unter *Forward Rendering* nach einiger Zeit zu einer thermisch bedingten Drosselung des *SoC*. Bei gedrosseltem SoC ist die Berechnung der *PBR-Shader* langsamer. Unter Verwendung von *Deferred Rendering* sind *Legacy*- und *PBR-Shader* gleich schnell. Jedoch drosselt Testgerät 2 in diesem Szenario bei beiden *Shadern*. Unter diesem *Renderpfad* lief die Anwendung im getesteten Szenario nicht flüssig.

5.7 Verhalten sich Geräte verschiedener Hersteller, die über dieselbe Hardware verfügen, in bestimmten Szenarien gleich?

Um dies zu überprüfen werden die Testgeräte 2 und 3 verwendet. Beide verfügen über dieselbe Hardware. Als *SoC* kommt der *Qualcomm Snapdragon 805* mit je 2,7 GHz zum Einsatz. Beide Geräte verfügen über 3 GB *RAM* und nutzen *Android* in der Version 5.1. Bei Testgerät 2 kommt das unangepasste *Android* von Google zum Einsatz. Testgerät 3 verwendet *Android* mit einer angepassten Benutzeroberfläche von Samsung. Die Bildschirmauflösung beträgt bei beiden Geräten 2560×1440 Pixel.

Bei Geräten gleicher Hardware ist zu erwarten, dass die Ergebnisse der Tests annähernd gleich sind. Um die Geräte miteinander zu vergleichen werden die bei der *Frame Time* ermittelten Abweichungen verschiedener Tests beider Geräte gegenübergestellt.

5.7.1 Testblock 1

Forward Rendering

In Abbildung 36 ist deutlich zu erkennen, dass Testgerät 2 unter *Forward Rendering* schneller ist als Testgerät 3. Bei Testgerät 3 gab es einen deutlichen Ausschlag der *Frame Time* bei Sekunde 22. Für die Berechnung der Abweichung zwischen beiden Testgeräten wurde dieser Wert durch einen gerundeten Wert, welcher das Mittel zwischen Sekunde 21 und 23 darstellt, ersetzt. Die *Frame Time* bei Testgerät 3 ist durchschnittlich um 7,23 ms über der von Testgerät 2.

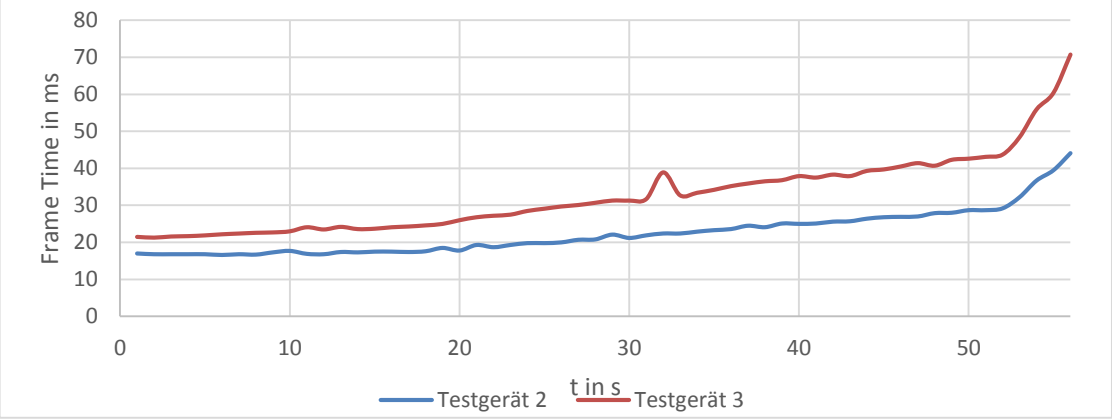

Abbildung 36: Vergleich zwischen Testgerät 2 und 3 bei Testblock 1 mit Forward Rendering

Deferred Rendering

Unter Verwendung von *Deferred Rendering* steigt die *Frame Time* bei Testgerät 2 stärker an als bei Testgerät 3. Dieser Sachverhalt wird in Abbildung 37 visualisiert. Die *Frame Time* liegt bei Testgerät 2 um durchschnittlich 21,73 ms über der von Testgerät 3.

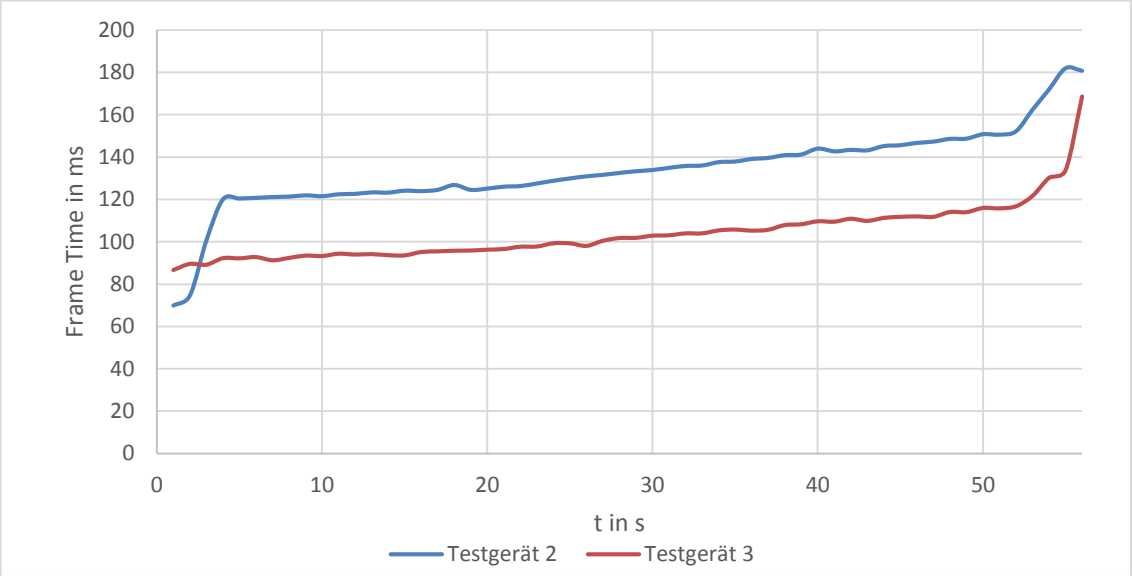

Abbildung 37: Vergleich zwischen Testgerät 2 und 3 bei Testblock 1 unter Deferred Rendering

In Testblock 1 gibt es deutliche Unterschiede zwischen beiden Geräten obwohl diese über dieselbe Hardware verfügen. Testgerät 2 liegt im *Forward Rendering* deutlich vor Testgerät 3. Unter *Deferred Rendering* brechen beide Geräte stark ein. Das zweite Testgerät bricht jedoch viel stärker ein als Testgerät 1. Bei Testgerät 2 beträgt der Unterschied zwischen *Forward Rendering* und *Deferred Rendering* im Durchschnitt 78,57 ms. Bei dem dritten Testgerät beträgt dieser nur 50,42 ms.

5.7.2 Testblock 3

In diesem Abschnitt werden die Testgeräte 2 und 3 auf Leistungsunterschiede bei der Berechnung von *PBR*-Shadern in *Forward Rendering* und *Deferred Rendering* analysiert.

Forward Rendering

Unter Verwendung der *PBR-Shader* unter *Forward Rendering* zeigt sich die temperaturbedingte-Drosselung von Testgerät 2. Abbildung 38 zeigt die Verläufe der *Frame Time* bei beiden Geräten. Es ist deutlich zu erkennen, dass die Werte der *Frame Time* von Testgerät 2 trotz der Drosselung fast immer niedriger sind als bei Testgerät 3. Im Durchschnitt berechnet Testgerät 2, trotz identischer Hardware, die Szene um 6,51 ms schneller. Der maximale Unterschied der *Frame Time* beider Geräte beträgt 46,17 ms während der minimale Unterschied bei 0,14 ms liegt. Sobald die *CPU* von Gerät 2 gedrosselt wird ist die *Frame Time* von Testgerät 3 kurzzeitig niedriger als die von Gerät 2. Dies tritt beispielsweise bei Sekunde 15 auf. Die *Frame Time* von Testgerät 2 liegt in diesem Messabschnitt um 1,1 ms oberhalb der des anderen Testgeräts.

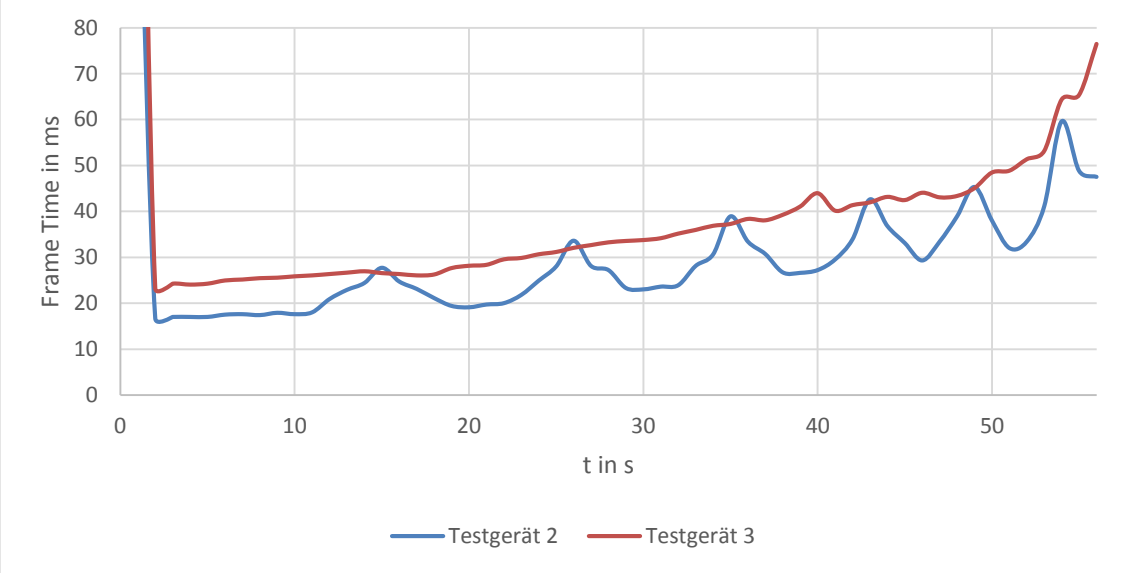

Abbildung 38: Vergleich zwischen Testgerät 2 und 3 bei Testblock 3 unter Forward Rendering

Deferred Rendering

Bei diesem *Renderpfad* ist Testgerät 2 langsamer als Testgerät 3. Abbildung 39 zeigt diesen Sachverhalt. Im Durchschnitt ist die *Frame Time* um 14,77 ms höher als die von Gerät 3. Minimal weichen die Daten um 0,42 ms und maximal um 23,62 ms voneinander ab.

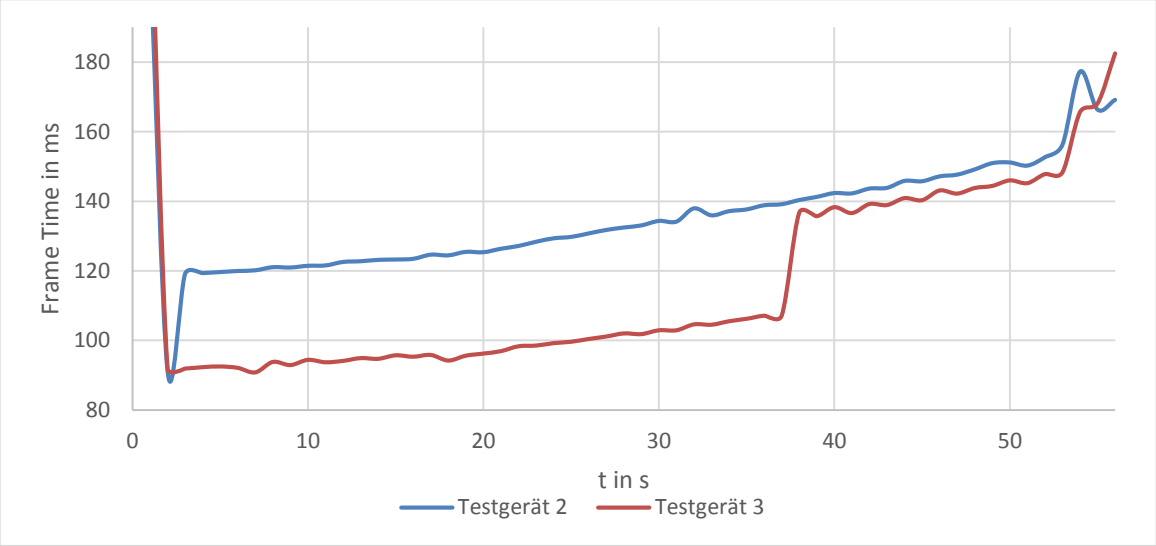

Abbildung 39: Vergleich zwischen Testgerät 2 und 3 bei Testblock 3 unter Deferred Rendering

Unter *Deferred Rendering* drosselt Testgerät 2 die *CPU*-Leistung von Anfang an. Zu Beginn der Aufzeichnungen liegt die *Frame Time* beider Geräte bei 91,3 ms für Testgerät 2 und 91,9 ms für Testgerät 3. Eine Sekunde später ist die *Frame Time* des zweiten Gerätes bereits um 27,3 ms höher, während die des anderen Testgerätes den Wert beibehält.

Im folgenden Auszug der *Logfile*[118] von Testgerät 2 ist die Drosselung der *CPU* nach der Ausgabe des ersten *Profiler*-Blocks zu sehen.

profiler_app_nexus_6_5_1_deferred.txt

09-14 15:29:15.919: I/ThermalEngine(1558): Mitigation:GPU[0]:500000000 Hz
(…)
09-14 15:29:16.917: D/Unity(20932): Android Unity internal profiler stats:
(…)
09-14 15:29:18.933: I/ThermalEngine(1558): Mitigation:GPU[0]:389000000 Hz

Anhand des Kurvenverlaufs in Abbildung 39 ist deutlich zu erkennen, dass es bei Testgerät bei Sekunde 38 einen Anstieg der *Frame Time* von Testgerät 3 gibt. Dieser wird wie bei Gerät 2 durch die Drosselung des *SoC* verursacht. Nachfolgender Auszug aus der *Logfile*[119] zeigt wie sich die Taktfrequenzen von *CPU* und *GPU* während der Ausführung von Testblock 3 verändern.

[118] Vgl. Anlage CD /profiler_app/profiler_app_nexus_6_5_1_deferred.txt
[119] Vgl. Anlage CD /profiler_app/profiler_app_note_4_5_1_deferred.txt

profiler_app_note_4_5_1_deferred.txt

```
09-23 10:06:46.687: D/U(979): limitCPUFreq:: freq = 1958400
(...)
09-23 10:06:46.687: D/U(979): limitGPUFreq:: freq = 389000000
(…)
09-23 10:12:50.057: D/U(979): limitCPUFreq:: freq = 1728000
(…)
09-23 10:12:50.067: D/U(979): limitGPUFreq:: freq = 300000000
```

5.7.3 Fazit

Bei den in diesem Test verglichenen *Android*-Geräten wurde erwartet, dass die Leistungsdaten ähnliche Ergebnisse liefern. Hierfür wurden das Testgerät 2 (*Motorola Nexus 6*) und 3 (*Samsung Galaxy Note 4*) miteinander verglichen. Beide Geräte nutzen die gleiche *SoC*, haben dieselbe Bildschirmauflösung und der Arbeitsspeicher fasst je 3 GB.

Die Analyse der aufgezeichneten Daten ergab, dass Testgerät temperaturbedingt die Leistung des *SoC* drosseln muss. Bei der Verwendung der *Legacy-Shader* und unter *Forward Rendering* macht sich diese Drosselung nicht bemerkbar. In diesem Fall ist die *Frame Time* geringer als bei Testgerät 3. Dies bedeutet, dass Testgerät 2 in unter diesen Bedingungen schneller ist.

Unter Verwendung der *PBR-Shader* unter *Forward Rendering* treten bei Testgerät 2 Drosselungen des *SoC* auf. Im durchgeführten Test war Testgerät 2 jedoch trotz der Drosselung im Durchschnitt schneller als Testgerät 3.

Sobald *Deferred Rendering* zum Einsatz kommt bricht die Leistung von Testgerät 2 stark ein. In diesen Tests wird der *SoC* des *Android*-Gerätes von Anfang an gedrosselt. Testgerät 3 ist in diesen Abschnitten deutlich schneller als Testgerät 2. Eine thermische Drosselung tritt jedoch auch bei dem *Samsung Galaxy Note 4* ein. In Testblock 3 der Analyseanwendung wurde der *SoC* des Gerätes nach 37 Sekunden gedrosselt.

Testgerät 2 liegt im *Forward Rendering*, in den analysierten Testblöcken, stets vor Nummer 3, obwohl es bei Verwendung der *PBR-Shader* den *SoC* drosseln muss. Unter *Deferred Rendering* ist das *Galaxy Note 4* in beiden analysierten Szenarien schneller als das *Motorola Nexus 6*. Die Anwendung läuft in diesen Szenarien jedoch auf keinem der beiden getesteten Geräte flüssig. Grund hierfür ist die *Frame Time*, die in jeder Sekunde, weit über 40 ms beträgt.

6 Fehlerbetrachtung

Während der Erstellung des Ablaufs zur Leistungsanalyse traten Fehler auf die in diesem Kapitel genauer betrachtet werden.

6.1 Datenverluste bei der Aufzeichnung

Die Verbindung des *Android*-Smartphones mit dem PC brach sporadisch ab. Dies resultierte darin, dass ein Test vollständig wiederholt werden musste.

Um einem vollständigen Datenverlust entgegenzuwirken, kann die Testanwendung in mehrere kleine Anwendungen aufgeteilt werden. Jeder Testblock oder jeder Abschnitt eines Testblocks kann als separate App ausgegeben werden. Daraus resultiert jedoch ein deutlich höherer Speicherverbrauch auf dem Smartphone, da die Anwendung mehrfach installiert werden muss um alle Tests abzubilden.

Auf älteren Smartphones kann die Aufzeichnung der Leistungsdaten sehr lange dauern. Beispielsweise dauerte die Aufzeichnung der Analyse-Anwendung auf einem *Samsung Galaxy S3* insgesamt 37 Minuten.[120] Zusätzlich werden sehr viele Daten via *ADB* empfangen. Dies kann bei einem zu niedrig eingestellten *Buffer* zu Datenverlusten führen. Auf einem *Samsung Galaxy S3* wurden 21.779 Zeilen mit *ADB* empfangen.

6.2 Speicherverbrauch im Android-Gerät

Hochaufgelöste 3D-Modelle können in einer sehr großen *Android-APK*-Datei resultieren. Bei zu geringem internem Speicher kann eine Anwendung unter Umständen nicht installiert werden. Um zu verhindern, dass eine sehr große *APK*-Datei installiert werden muss, können 3D-Modelle in Unity komprimiert werden. Diese Kompression des 3D-Modells reduziert die Dateigröße der App. Die Modelle werden zur Laufzeit der App wieder dekomprimiert. Die Kompression hat keine Auswirkung auf den Speicherverbrauch während die App ausgeführt wird.[121]

---

[120] Vgl. Anlage CD /profiler_app/profiler_app_galaxy_s3_4_4_4_forward.txt

[121] Vgl. UNITY, https://docs.unity3d.com/355/Documentation/Manual/ReducingFilesize.html

Um zu überprüfen, ob die Kompression die Testergebnisse verfälscht, wurden die ersten beiden Testblöcke, der Testanwendung, dreimal durchlaufen. Der erste Durchgang erfolgte ohne Kompression.[122] Im zweiten mit mittlerer und im dritten mit hoher Kompression der 3D-Modelle.[123][124] Ohne Kompression betrug die *App*-Größe 210.983 MB. Bei mittlerer sank der Speicherverbrauch auf 96,579 MB und bei hoher auf 78,534 MB.

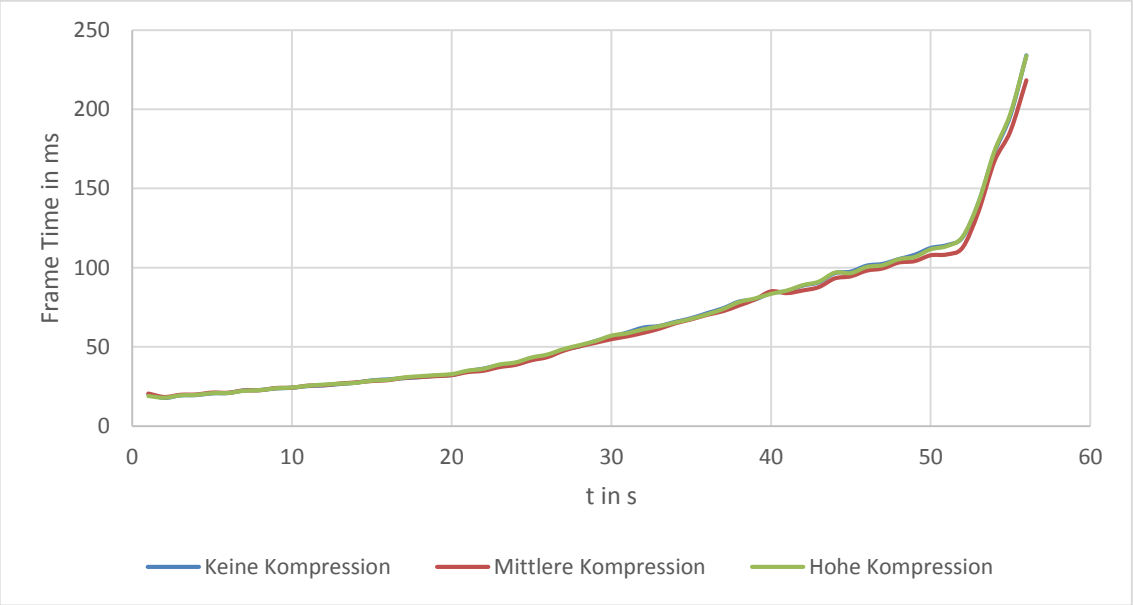

Abbildung 40: Frame Time Verläufe mit verschiedenen Kompressionsarten[125][126][127]

Über die Standardabweichung der *Frame Time* wurde überprüft ob es Auswirkungen auf die Anwendungsleistung gibt. Der Mittelwert der Abweichung beträgt 1,128 ms. Dies entspricht einem niedrigeren Wert als der in Kapitel 0 ermittelten Schwankung zwischen den einzelnen Anwendungsdurchläufen. Abbildung 40 zeigt die annähernd gleichen Verläufe der *Frame Time* bei unterschiedlichen Kompressionsarten.

Aus den Tests mit unterschiedlichen Kompressionsstärken resultiert, dass es keine Verfälschung der Testergebnisse durch die Kompression von 3D-Modellen gibt. Die Anwendungsgröße reduziert sich jedoch signifikant. Dies kommt vor allem *Android*-Geräten mit geringem internem Speicher zugute.

---

[122] Anlage CD /apps/profiler_app_uncompressed.apk
[123] Anlage CD /apps/profiler_app_medium_compression.apk
[124] Anlage CD /apps/profiler_app_high_compression.apk
[125] Vgl. Anlage CD /profiler_app_compression/profiler_app_no_compression.txt
[126] Vgl. Anlage CD /profiler_app_compression/profiler_app_medium_compression.txt
[127] Vgl. Anlage CD /profiler_app_compression/profiler_app_high_compression.txt

6.3 Abweichungen bei der Anzahl der Triangles

Die in den *Unity-Profiler-Logfiles* gezeigte Anzahl der berechneten *Triangles* weicht stark von der eigentlichen *Triangles* der 3D-Modelle ab. Bei hoch aufgelösten Modellen ist der Unterschied noch stärker.

Hoch aufgelöste 3D-Modelle werden in Unity in kleinere Modelle unterteilt sobald die Anzahl der Vertices einen Grenzwert übersteigt. Der Import eines 3D Modells mit mehr als 65534 Vertices resultiert in folgender Meldung:

"Meshes may not have more than 65534 vertices or triangles at the moment"

Sobald ein einzelnes 3D-Modell diese Zahl übersteigt, wird es automatisch in 2 oder mehr Modelle unterteilt. Die Unterteilung kann nicht beeinflusst werden. Grund für die Grenze von 65534 Vertices ist die Verwendung eines 16-Bit-*Integer*-Wertes für die Speicherung von 3D-Modellen.

Die Unterteilung von 3D-Modellen resultiert in erhöhten *Draw Call*-Zahlen, obwohl visuell nur ein Objekt sichtbar ist. Abbildung 41 visualisiert den Anstieg der *Draw Calls* durch den Anstieg der *Triangles* des Charaktermodells. Die Anzahl der Objekte bleibt konstant. Jedoch unterteilt Unity die höher aufgelösten 3D-Modelle selbstständig in mehrere kleinere Objekte, welche in zusätzlichen *Draw Calls* resultieren.

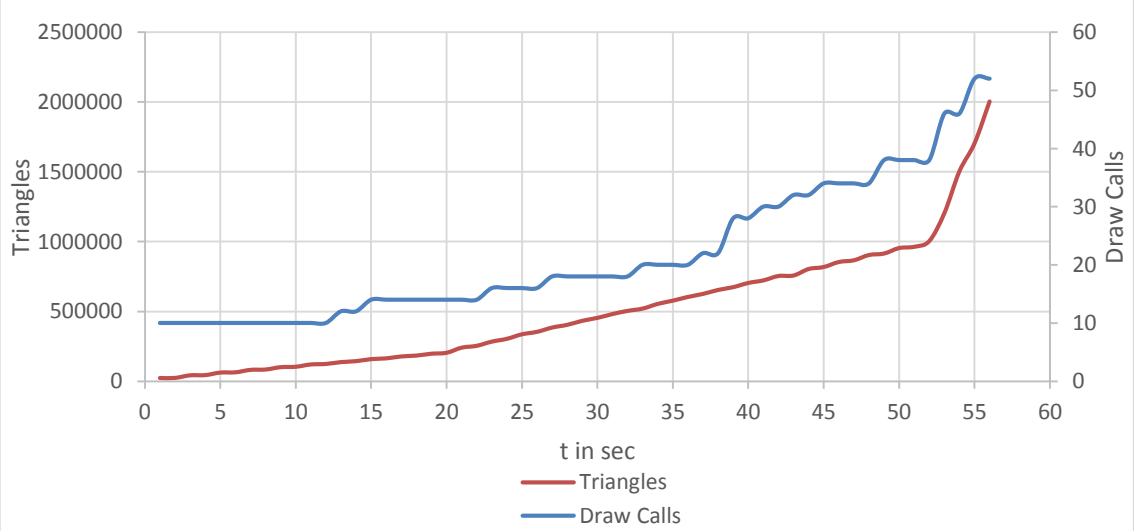

Abbildung 41: Draw Call-Anstieg durch Erhöhung der Triangles[128]

[128] Vgl. Anlage CD /profiler_app/ profiler_app_galaxy_s3_4_4_4_forward.txt

Zusätzlich gibt es einen Unterschied zwischen der Anzahl der *Vertices* und *Triangles* des Models und derer die in den *Logfiles* verzeichnet sind. Innerhalb der *Unity-Profiler*-Daten wird die Anzahl der *Triangles* und *Vertices* anhand der Anzahl jener angegeben, die gerendert werden müssen. Die Anzahl der gerenderten *Triangles* und *Vertices* hängt von verschiedenen Faktoren ab. Folgender Eintrag aus der *Unity*-Dokumentation erklärt diesen Sachverhalt:

"What you are looking at is the number of vertices/triangles actually being sent to the GPU for rendering. In addition to the case where the material requires them to be sent twice, other things like hard-normals and non-contiguous UVs increase vertex/triangle counts significantly compared to what a modeling app tells you. Triangles need to be contiguous in both 3D and UV space to form a strip, so when you have UV seams, degenerate triangles have to be made to form strips - this bumps up the count."[129]

Aus diesem Grund ist die im *Profiler* angezeigte Zahl der *Triangles* und *Vertices* immer höher als die Zahl beim Import der 3D-Modelle.

6.4 Schattendarstellung im Rendering

In Kapitel 5.4 wurden Verglichen wie sich Lichtquellen auf die 3D-Leistung auswirken. Ein Ergebnis der Tests war, dass unter *Forward Rendering* keine sichtbaren Schatten gerendert wurden. In der Szene wurde eine steigende Anzahl von Punktlichtern geladen. Diese Lichtquellen können, unter *Forward Rendering*, in *Unity* auf mobilen Endgeräten keine Schatten werfen. Die einzigen unterstützten Lichtquellen sind *Directional Lights*. Folgender Auszug aus der Unity Dokumentation verdeutlicht dies:

„Forward rendering path supports only one directional shadow casting light."[130]

Unter *Deferred Rendering* wurden die Schatten aller Lichtquellen dargestellt. Jedoch gab es keinen Unterschied zwischen harten und weichen Schatten. Auf mobilen Endgeräten ist es nicht möglich weiche Schatten darzustellen:

„Shadows on Android and iOS have these limitations: soft shadows are not available"[131]

---

[129] UNITY, http://docs.unity3d.com/Manual/class-Mesh.html, 25.10.15

[130] UNITY, http://docs.unity3d.com/460/Documentation/Manual/Shadows.html, 25.10.15

[131] UNITY, http://docs.unity3d.com/Manual/TroubleShooting.html#AndroidTroubleShooting, 25.10.15

7 Zusammenfassung

Ziel der Arbeit war es einen Analyseprozess zur Optimierung von mobilen 3D-Anwendungen zu entwickeln. Die Analyse der Leistungsdaten beschränkte sich auf *Android*-Geräte. *iOS*-Geräte wurden nicht analysiert. Um Apps für dieses mobile Betriebssystem zu entwickeln wird ein Apple-Computer mit *Mac OS* vorausgesetzt über den *PiXABLE STUDIOS* nicht verfügen. Auf eine Analyse von *Windows Phone*-Geräten wurde aufgrund des geringen Marktanteils verzichtet.

Die Analyse der *Android*-Geräte wurde bei bestehender USB-Verbindung mit einem PC durchgeführt. Dabei wurde auf das *Android-Debug*-Werkzeug *DDMS* zurückgegriffen. Als Schnittstelle zwischen dem mobilen Endgerät und dem PC diente *ADB*. Die zu testenden *Android*-Geräte mussten, zur Leistungsanalyse, in den Entwicklermodus geschalten werden. Nur über diesen ist ein *Debugging* über die *ADB*-Schnittstelle möglich. Über diese Schnittstellen wurden die Leistungsdaten eines *Android*-Gerätes über einen PC aufgezeichnet. Die aufgezeichneten Leistungsdaten wurden, für die spätere statistische Auswertung, als Textdokumente auf dem PC gespeichert.

Die Aufzeichnung der Daten über *ADB* bringt Einschränkungen mit sich. Es können nur Geräte getestet werden bei denen eine USB- oder WLAN-Verbindung mit einem PC besteht. Denn die *Profiler*-Daten können nicht direkt auf dem *Android*-Gerät gespeichert werden. Entsprechend ist ein nachträgliches Abrufen dieser Daten nicht möglich. Dies macht eine Distribution der App zur schnellen Leistungserfassung vieler Geräte schwierig. Denn der komplette Testaufbau mit *ADB*-Verbindung und *DDMS* müsste für jede neue Testumgebung aufgebaut werden.

Zunächst wurde geprüft, ob eine Leistungsanalyse von mobilen Endgeräten möglich ist. Hierfür wurde eine App mit *Unity* erstellt und auf einem *Android*-Gerät ausgeführt. Die App gliederte sich in mehrere Testszenen. In der ersten Testszene wurden simple geometrische-Körper nacheinander eingeblendet. Die zweite Szene zeigte nacheinander verschiedene 3D-Charaktermodelle. Die dritte Szene bestand aus einer von *PiXABLE* erstellten *PLAYMOBIL*-3D-Umgebung. Mit dieser App wurde überprüft, ob statistische-Daten von einem *Android*-Gerät aufgenommen werden können. Die Steuerung der App erfolgt innerhalb von *Unity* über *Skripte* in *C#*. Diese ermöglichen die Automatisierung von Objekt- und Szenenwechseln innerhalb der App. Hierdurch ist es möglich, die Ausführung unterschiedlicher Szenarien zu automatisieren.

Innerhalb der Tests wurde zunächst geprüft, ob die Daten der Testanwendung statistisch auswertbar sind. Zunächst wurde geprüft, ob alle Daten des *Unity-Profilers* über *ADB* aufgezeichnet werden. Dabei ergab sich, dass bei starker Auslastung des *Android*-Gerätes durch die App die Leistungsdaten fehlerhaft erfasst werden. Grund hierfür war eine Abweichung von der *Frame Rate* der *Android*-Geräte, welche ab *Android* 4.1 bei 60 fps liegt. Um dieses Problem zu beheben wurde ein *Rendering* von 60 fps erzwungen. Dies hatte zur Folge, dass die Anwendung bei leistungsintensiven Tests in Zeitlupe ablief. Dadurch erhöhten sich die Aufzeichnungszeiten der Leistungsdaten stark. Bei einem *Samsung Galaxy S3* dauerte die Aufzeichnung, der 8 Minuten und 34 Sekunden dauernde Analyseanwendung, somit 37 Minuten. Mit der erzwungenen Berechnung von 60 fps wurden alle Daten fehlerfrei aufgezeichnet. Es wurden weitere Tests mit verschiedenen *Frame Rates* durchgeführt. Hierbei wurden vielfache von 60 fps getestet. Bei einer Verdoppelung der *Frame Rate* auf 120 fps verdoppelte sich auch die Aufzeichnungszeit in komplexen Szenen. Es erhöhte sich jedoch auch die Genauigkeit der Daten, da pro Sekunde 2 *Profiler-Blöcke* aufgezeichnet wurden. Bei einer halbierten *Frame Rate* von 30 fps wurden *Profiler*-Blöcke nur im Abstand von 2 Sekunden übertragen. Hierdurch kann die Aufzeichnungszeit gerade bei langsamen Testgeräten signifikant verkürzt werden. Jedoch verringert dies die Genauigkeit der Daten gerade bei sich schnell ändernden 3D-Umgebungen.

Neben der Übertragung aller *Profiler*-Daten war es ebenfalls wichtig, dass die Daten reproduzierbar sind. Entsprechend musste geprüft werden, ob bei mehreren durchgeführten Tests dieselben Ergebnisse zustande kommen. Hierfür wurden mehrere Testreihen aufgezeichnet. Vergleiche der Testreihen ergaben, dass jeder Aufzeichnungsdurchgang annähernd gleiche Ergebnisse lieferte. Jedoch erhöhte sich die Standardabweichung, der *Frame Time* zwischen mehreren Testdurchläufen, mit steigender Komplexität der 3D-Umgebungen. Bei der ersten der drei Testszenen betrug sie im Durchschnitt 0,01 ms. In der dritten und gleichzeitig komplexesten Szene der Testanwendung, betrug sie 3,48 ms.

Der zur Aufzeichnung der Daten verwendete PC beeinflusste die Leistungserfassung nicht. Hierfür wurden jeweils 10 Datensätze der Testanwendung auf drei unterschiedlichen Computern aufgezeichnet. Die Standardabweichung der Leistungsdaten der unterschiedlichen Aufzeichnungssysteme bewegte sich im bereits vorher ermittelten Rahmen.

Eine direkte Auswertung der aufgezeichneten Leistungsdaten war nicht möglich. Um sie statistisch analysieren zu können mussten die Daten umgewandelt werden. Diese Umwandlung erfolgt über, ein für diese Aufgabe erstelltes, *Python-Skript*. Dieses liest die *Unity-Logfiles*, filtert irrelevante Daten heraus und speichert alle Leistungsdaten der App in eine Datenbank. Abschließend wird die Datenbank als *CSV*-Datei ausgegeben. Die, in dieser Datei gespeicherten Daten, können mithilfe von Tabellenkalkulationsprogrammen geöffnet und ausgewertet werden.

Die Umwandlung der Daten läuft automatisiert ab. Die *Python*-Laufzeitumgebung muss hierfür installiert sein. Das für die Umwandlung erstellte *Skript* nutzt zwei Eingabevariablen. Diese sind das *Skript*, sowie die umzuwandelnde *Logfile*. Über eine zusätzliche *Batch*-Datei wurde der Prozess weiter automatisiert. Diese ermöglicht es mehrere *Logfiles* nacheinander umzuwandeln.

Ein Ziel dieser Arbeit war es herauszufinden ob sich Parameter identifizieren lassen, welche die Leistung auf bestimmten Geräten negativ beeinflussen. Hierzu wurde mittels vordefinierter Forschungsfragen eine App für die Leistungsanalyse erstellt. Der Verlauf dieser App wurde auf mehreren unterschiedlichen *Android*-Geräten aufgezeichnet. Sich negativ auswirkende Parameter wurden mit Hilfe der statistischen Analyse ermittelt. Es zeigte sich beispielsweise, dass Geräte unterschiedlicher Hersteller, in diesem Fall *Samsung* und *Motorola*, trotz gleicher Hardware unterschiedliche 3D-Leistung besitzen. Im Falle der beiden verglichenen Geräte war dies auf eine thermische-Drosselung des *Motorola*-Gerätes zurückzuführen. Ein weiteres Ergebnis der Auswertung ist, dass sich moderne *PBR-Shader* auf mobilen Geräten zwar ausführen lassen, die 3D-Leistung jedoch bei komplexen Szenen signifikant schlechter ist als unter Verwendung der *Legacy-Shader*.

8 Fazit

Durch die Leistungserfassung *Android*-Endgeräte ist es möglich die App-Entwicklung zu optimieren. Faktoren, welche die Leistungsfähigkeit bestimmter *Android*-Geräte beeinflussen konnten auf den getesteten Geräten identifiziert werden. Im Gegensatz zu herkömmlichen Leistungsanalyse-Apps konnten mit Hilfe des Analyseprozesses Faktoren identifiziert werden, welche die Leistungsfähigkeit bestimmter *Android*-Geräte beeinflussen. Anhand dieser ist es möglich, eine App so zu optimieren, dass sie auf bestimmten Geräten bessere Leistungen erzielt.

Der Versuchsaufbau, der für die Leistungserfassung nötig ist, ermöglicht es, beliebige *Android*-Geräte zu analysieren. Von Nachteil ist jedoch dessen Komplexität. Während Apps wie *3DMARK* über die App die aufgezeichneten Daten an einen Webserver weitergeben, können die über *ADB* ausgegebenen Daten nur über eine bestehende USB- oder WLAN-Verbindung mit einem PC empfangen und gespeichert werden. Deshalb gestaltet sich eine Distribution der Analyseanwendung, über Vertriebsplattformen wie *Google Play,* aufgrund der Komplexität des Versuchsaufbaus als schwierig. Nutzer, die ihr Gerät für eine Leistungsanalyse bereitstellen, müssten in den Prozess der Leistungsanalyse eingewiesen werden, da die Daten über die *ADB*-Schnittstelle aufgezeichnet werden müssen. Dies sorgt dafür, dass die Datenerhebung sehr aufwendig ist, da jeder Teilnehmer, der zur Leistungserfassung beitragen möchte, muss alle für die Aufzeichnung benötigte Hard- und Software besitzen. Außerdem müssten sich etwaige Dritte bereiterklären ihre Geräte für den Test in den Entwicklermodus zu versetzen, denn nur in diesem Modus ist die Ausgabe der Leistungsdaten möglich.

Für *PiXABLE STUDIOS* ist der Analyseablauf nützlich. Mit den gewonnenen Erkenntnissen lassen sich Leistungsdefizite von Geräten vor oder während der Entwicklung von Apps erfassen. Dies hat besonders bei der Entwicklung von internen Anwendungen von Firmenkunden Vorteile, da diese explizite Anforderungen an die Geräte stellen, auf denen die Apps laufen sollen. Mithilfe des in dieser Masterarbeit entwickelten Analyseablaufs ist es für *PiXABLE* möglich Anwendungen optimal an die Geräte anzupassen.

Literaturverzeichnis

3DS MAX 2015 SAMPLE FILES. URL: https://knowledge.autodesk.com/support/3ds-max/downloads/caas/downloads/content/3ds-max-2015-sample-files.html (Stand: 05.10.15)

ANDROID DEVELOPER. URL: http://developer.android.com/ (Stand: 01.11.15)

APPLE DEVELOPER: App Programming Guide for iOS. Apple Inc. Cupertino 2015. URL: https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf/ (Stand: 02.12.15)

BIRN Jeremy: Lightning & Rendering. Addison-Wesley. München 2007.

BRUNKEN Yannick: Was ist Drag and Drop? Einfach erklärt. 2015. URL: http://praxistipps.chip.de/was-ist-drag-and-drop-einfach-erklaert_41228 (Stand: 01.11.15)

CHIN Danton u.a.: More iPhone Cool Projects: Cool Developers Reveal the Details of Their Cooler Apps and Discuss Their iPad Development Experiences. Springer Scient+Business Media. New York 2010.

DROIDWIKI. URL: https://www.droidwiki.de/ (Stand: 01.11.15)

FLAVELL Lance: Beginning Blender: Open Source 3D Modeling, Animation, and Game Design. Springer Science+Business Media. New York 2010.

FUTUREMARK. URL: http://www.futuremark.com/ (Stand: 31.10.15)

GERLICH Rainer, Gerlich Ralf: 111 Thesen zur erfolgreichen Softwareentwicklung: Argumente und Entscheidungshilfen für Manager. Konzepte und Anleitungen für Praktiker. Springer-Verlag. 2005.

GREENBERG Ira: Processing: Creative Coding and Computational Art. Springer-Verlag. New York 2007.

HETLAND Magnus Lie: Beginning Python: From Novice to Professional, Second Edition. Apress Media LLC. New York 2008.

IHLENFELD Jens: JELLY BEAN: Project Butter macht Android 4.1 deutlich schneller. 2012. URL: http://www.golem.de/news/jelly-bean-project-butter-macht-android-4-1-deutlich-schneller-1206-92806.html (Stand: 01.11.15)

JAVA. URL: https://www.java.com/ (Stand: 30.11.15)

KRAJCI Iggy, Cummings Darren: Android on x86: An Intoduction to Optimizing for Intel Architecture. Apress Media LLC. New York 2014.

MALIZIA Alessio: Mobile 3D Graphics. Springer-Verlag. London 2006.

MASTERSOLUTION: Mastersolution AG übernimmt PiXABLE STUDIOS. 2014. URL: http://blog.mastersolution.ag/2014/03/06/mastersolution-ag-ubernimmt-pixable-studios (Stand: 01.11.15)

NISCHWITZ Alfred u.a.: Computergrafik und Bildverarbeitung. Vieweg + Teubner. 2011.

OWENS Brent: Forward Rendering vs. Deferred Rendering. 2013. URL: http://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342 (Stand: 01.11.15)

PiXABLE STUDIOS. URL: http://pixablestudios.com/ (Stand 01.11.15)

PYTHON. URL: https://www.python.org (Stand: 30.11.15)

PYTHON DOCUMENTATION. URL: https://docs.python.org/ (Stand: 30.11.15)

REIS Ricardo, Lubaszewski Marcelo, Jess Jochen A.G.: Design of Systems on a Chip: Design and Test. Springer. 2006.

SAMSUNG DEVELOPER: Samsung Android USB Driver for Windows. 2015. URL: http://developer.samsung.com/technical-doc/view.do?v=T000000117 (Stand: 01.10.15)

SCHLIMM Niklas u.a.: Performance-Analyse und –Optimierung in der Softwareentwicklung: Springer-Verlag. 2007.

SMITH Brian L.: Foundation 3ds Max 8 Architectural Visualization. Springer-Verlag. New York 2006.

STATISTA: Marktanteile der mobilen Betriebssysteme am Absatz von Smartphones in Deutschland von Januar 2012 bis September 2015. URL: http://de.statista.com/statistik/daten/studie/225381/umfrage/marktanteile-der-betriebssysteme-am-smartphone-absatz-in-deutschland-zeitreihe/ (Stand: 01.11.15)

TÖNNIS Marcus: Augmented Reality: Einblicke in die Erweiterte Realität. Springer-Verlag. Berlin Heidelberg 2010.

UNITY3D. URL: https://unity3d.com/ (Stand: 01.11.15)

UNITY Technologies: Unity Manual. URL: http://docs.unity3d.com/Manual/index.html. 2015 (Stand: 01.11.15)

WILSON Joe: Physically Based Rendering, And You Can Too! URL: http://www.marmoset.co/toolbag/learn/pbr-practice (Stand: 01.11.15)

WITTE Jörg: Programmieren in C#: Von den ersten Gehversuchen bis zu den Sieben-Meilen-Stiefeln. Teubner Verlag. Wiesbaden 2004.

Anlagen

Anlage 1 **Change_Scene.cs**

```
using UnityEngine;
using System.Collections;

public class Change_Scene : MonoBehaviour {

public float time;
public string next;



IEnumerator Start ()
        {
        yield return new WaitForSeconds (time);
        yield return new WaitForFixedUpdate();
        Debug.Log ("changing level to: " + next);
        Application.LoadLevel (next);
        }
}
```

Anlage 2 **counter_v01.py**

```
print "\nunity logfile counter", "\n"
import sys

input_file = sys.argv[1]

with open(input_file) as logfile:
    datablocks = 0
    for line in logfile:
        current_line = line.rstrip()
        if "Android Unity internal profiler stats:" in current_line:
            print current_line
            datablocks = datablocks +1
        if "changing level to:" in current_line:
            print "scenechange"

print "processed datablocks: ", datablocks
```

Anlage 3 **counter_v02.py**

```
print "\nunity logfile counter", "\n"
import sys

input_file = sys.argv[1]

with open(input_file) as logfile:
    datablocks = 0
    for line in logfile:
        current_line = line.rstrip()
        if "changing level to:" in current_line:
            print "scenechange - processed datablocks: ", datablocks
        if "Android Unity internal profiler stats:" in current_line:
            print current_line
            datablocks = datablocks +1
        if "switching object" in current_line:
            print "objectchange - processed datablocks: ", datablocks


print "processed datablocks: ", datablocks
```

Anlage 4 **convert_logfiles.bat**

```
@ for %%i in (%*) do "c:\python27\python.exe" filereader_v17.py %%i

cmd /k
```

Anlage 5 **filereader_v17.py**

```
print "\nunity logfile converter", "\n"
#converts unity debug logfiles from adb to excel readable csv files
#csv files are separated by tabs
#eric schubert
#08/2015

#import functions for recompile, argument input and os pathnames
import re, sys, os, collections
frametime = 0
#define second file argument as the input file
#output file name gets generated from input file name
#changes output file format to csv
input_file = sys.argv[1]
input_file = os.path.basename(input_file)
print "opened logfile as " + input_file
output_file = os.path.basename(input_file.replace(".txt","_converted.csv"))

#define dictionary
#dictionary with subdictionarys, each dictionary contains a list of values
#ordered dictionary with ordered subdictionaries is created by a function
def create_dict(my_dict):
    print "creating dict"
    my_dict['frametime']=collections.OrderedDict()
    my_dict['frametime']['min']=[]
    my_dict['frametime']['max']=[]
    my_dict['frametime']['avg']=[]
    my_dict['draw calls']=collections.OrderedDict()
    my_dict['draw calls']['min']=[]
    my_dict['draw calls']['max']=[]
    my_dict['draw calls']['avg']=[]
    my_dict['verts']=collections.OrderedDict()
    my_dict['verts']['min']=[]
    my_dict['verts']['max']=[]
    my_dict['verts']['avg']=[]
    my_dict['tris']=collections.OrderedDict()
```

```
my_dict['tris']['min']=[]
my_dict['tris']['max']=[]
my_dict['tris']['avg']=[]
my_dict['batches']=collections.OrderedDict()
my_dict['batches']['min']=[]
my_dict['batches']['max']=[]
my_dict['batches']['avg']=[]
my_dict['player-detail']=collections.OrderedDict()
my_dict['player-detail']['physx']=[]
my_dict['player-detail']['animation']=[]
my_dict['player-detail']['culling']=[]
my_dict['player-detail']['skinning']=[]
my_dict['player-detail']['batching']=[]
my_dict['player-detail']['render']=[]
my_dict['player-detail']['fixed-update-count']=[]
my_dict['static batching']=collections.OrderedDict()
my_dict['static batching']['batched draw calls']=[]
my_dict['static batching']['batches']=[]
my_dict['static batching']['verts']=[]
my_dict['static batching']['tris']=[]
my_dict['dynamic batching']=collections.OrderedDict()
my_dict['dynamic batching']['batched draw calls']=[]
my_dict['dynamic batching']['batches']=[]
my_dict['dynamic batching']['verts']=[]
my_dict['dynamic batching']['tris']=[]
my_dict['cpu-player']=collections.OrderedDict()
my_dict['cpu-player']['min']=[]
my_dict['cpu-player']['max']=[]
my_dict['cpu-player']['avg']=[]
my_dict['cpu-ogles-drv']=collections.OrderedDict()
my_dict['cpu-ogles-drv']['min']=[]
my_dict['cpu-ogles-drv']['max']=[]
my_dict['cpu-ogles-drv']['avg']=[]
my_dict['cpu-present']=collections.OrderedDict()
my_dict['cpu-present']['min']=[]
my_dict['cpu-present']['max']=[]
my_dict['cpu-present']['avg']=[]
```

```
    my_dict['mono-memory']=collections.OrderedDict()
    my_dict['mono-memory']['used heap']=[]
    my_dict['mono-memory']['allocated heap']=[]
    my_dict['mono-memory']['max number of collections']=[]
    my_dict['mono-memory']['collection total duration']=[]
    my_dict['mono-scripts']=collections.OrderedDict()
    my_dict['mono-scripts']['update']=[]
    my_dict['mono-scripts']['fixedUpdate']=[]
    my_dict['mono-scripts']['coroutines']=[]
my_dict=collections.OrderedDict()
create_dict(my_dict)

#recompile function to clean up the values
pattern = re.compile("\s+([a-zA-Z\s-]+):?\s+((-*[0-9]+\s..\s-*[0-9]+)|(-*[0-9]+\.*-*[0-9]*))")
key_name_pattern = re.compile("(.+:) (.+)")

#function to write the dictionary
def process_line(current_line, my_dict):
    key_value_list=current_line.lstrip().split(">")
    key_name = key_value_list[0]
    key_name_list = key_name_pattern.findall(key_name)
    for elements in key_name_list:
        key_name = elements[1]
    key_values = key_value_list[1]
    match = pattern.findall(key_values)
    if key_name not in my_dict:
        return False
    for each_group in match:
        if each_group not in match:
            return False
        sub_key_name = each_group[0]
        sub_key_name = sub_key_name.rstrip()
        #write subkey values to the subdictionary
        sub_key_value = each_group[1]
        #if "frametime" in current_line:
        #    if sub_key_value == str(0.0):
        #        sub_key_value = sub_key_value
```

```
        #    else:
        #        fps = 1000 / float(sub_key_value)
        #        fps = round(fps, 2)
        #        sub_key_value = str(fps)
        if ".." in sub_key_value:
            #checks if 2 values exist, happens in fixed update count
            #saves only the second value to the dictionary
            sub_key_value = sub_key_value.split(" .. ")[1]
            my_dict[key_name][sub_key_name].append(sub_key_value)
        else:
            my_dict[key_name][sub_key_name].append(sub_key_value)

#function to save the dictionary to a csv file
def dict_to_csv(output_file, my_dict):
    with open(output_file, "w") as converted_csv:
        #get the first header line from the main key names
        csv_headers = my_dict.keys()
        key_number = 0
        for key in csv_headers:
            key_number = key_number +1
            if key in my_dict:
                #sets subkey counter to zero for every key block
                count = 0
                for value in my_dict[key]:
                    #count the amount of subkeys to define the spacing
                    count = count +1
                #print key
                #print count
                insert = 1
                while insert < count:
                    #insert spacing to the csv_headers list at the current key position
                    csv_headers.insert(key_number, "---")
                    insert = insert + 1
            else:
                False
        #creation of subheaders list
        csv_subheaders=[]
```

```
for k, v in my_dict.iteritems():
    for subk, subv in v.iteritems():
        #gets the subheader name from the subk of the nested dictionary
        #appends the subk at the end of the subheaders list
        csv_subheaders.append(subk)
#writes headers and subheaders
converted_csv.write("\t")
for header in csv_headers:
    converted_csv.write(header + "\t")
converted_csv.write("\n" + "Data" +"\t")
for subheader in csv_subheaders:
    converted_csv.write(subheader + "\t")
converted_csv.write("\n")
#creation of empty values list
list_subv = []
for k, v in my_dict.iteritems():
    for subk, subv in v.iteritems():
        #appends lists of values so that there is a list of multiple lists
        list_subv.append(subv)
#remaps the list of lists
#values that are side by side in parallel lists get one list per row
#these rows can be written by the write function
list_subv = map(None,*list_subv)
#counter value of 1 for the current datablock
counter = 1
for subv in list_subv:
    #writes the current datablock number and increases the counter
    #tabstop as a seperator between the values

    if "----------" in subv:
        converted_csv.write("---" + "\t")
        counter = counter + 0
    else:
        converted_csv.write(str(counter) + "\t")
        counter = counter +1
    for value in subv:
        if value is None:
```

```
                #if there is a none value it saves it to the csv
                converted_csv.write(str(value) + "\t")
            else:
                #if there is a numerical value with dot it converts the dot to comma and saves it to the csv
                #with the comma the values get interpreted as numeric in the german excel version
                converted_csv.write(str(value.replace(".", ",")) + "\t")
        converted_csv.write("\n")
    #prints the processed datablocks and closes the csv
    print "\nprocessed", scenecount, "scenes"
    print "processed", counter -1, "values \n"
    converted_csv.close()
print "saved csv as " + output_file

#open the unity profiler logfile and call function line by line

with open(input_file) as logfile:
    scenecount = 1
    for line in logfile:
        current_line = line.rstrip()
        if "changing level to:" in current_line:
            scenecount = scenecount +1
            for k, v in my_dict.items():
                for subk, subv in v.items():
                    subv.append("----------")
        else:
            if ">" in current_line:
                process_line(current_line, my_dict)

#run the csv file writer function after the dict is created
dict_to_csv(output_file, my_dict)
```

Anlage 6 **Load_Objects.cs**

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Load_Objects : MonoBehaviour {

	public float time = 5.0f;
	public List<GameObject> object_list = new List<GameObject>();
	public GameObject object_list_holder;
	private int object_count;

	IEnumerator Start ()
	{
		//object_count=object_list.Count;
		//Debug.Log ("objects in list " + object_count);

		foreach(Transform child in object_list_holder.transform)
		{
			object_list.Add (child.gameObject);
			//Debug.Log (child.gameObject);
		}
		Debug.Log (object_list);
		yield return new WaitForFixedUpdate();

		foreach(GameObject obj in object_list)
		{
			Debug.Log (obj);
			obj.SetActive (true);
			yield return new WaitForSeconds (time);
			yield return new WaitForFixedUpdate();
			Debug.Log ("Loading object");
		}
	}
}
```

Anlage 7 **Profiler_Framerate.cs**

```
using UnityEngine;
using System.Collections;

public class Profiler_Framerate : MonoBehaviour {
public int framerate = 60;
	void Start(){
		Time.captureFramerate = framerate;
	}
}
```

Anlage 8 **Quit.cs**

```
using UnityEngine;
using System.Collections;

public class Quit : MonoBehaviour {

public float time;

IEnumerator Start ()
        {
        yield return new WaitForSeconds (time);
        yield return new WaitForFixedUpdate();
        Application.Quit ();
        }
}
```

Anlage 9 **Switch_Objects.cs**

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Switch_Objects : MonoBehaviour {

	public float time = 5.0f;
	public List<GameObject> object_list = new List<GameObject>();
	private int object_count;

	IEnumerator Start ()
	{
		object_count=object_list.Count;
		Debug.Log ("objects in list " + object_count);
		yield return new WaitForFixedUpdate();

		foreach(GameObject obj in object_list)
		{
			Debug.Log (obj);
			obj.SetActive (true);
			yield return new WaitForSeconds (time);
			obj.SetActive (false);
			yield return new WaitForFixedUpdate();
			Debug.Log ("switching object");

		}
	}
}
```

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Dresden, 15. Dezember 2015 Eric Schubert